PREV
About the Author

NEXT
Chapter 2 Create and use types

CHAPTER 1
Manage program flow

This chapter is focused on the way that programs run inside your computer. In the early days of computing this was a very simple process, with a single program executing on a single *Central Processing Unit* (CPU). Modern applications, however, are not implemented by an individual program following a single sequence of instructions. Today's applications typically contain large numbers of cooperating processes. There are two reasons why this is the case.

IMPORTANT
Have you read page xix?

It contains valuable information regarding the skills you need to pass the exam.

First, adding additional CPU elements increases the amount of data that can be processed in a given time. In the same way that two cars can carry twice as many passengers as one, adding a second CPU will increase the amount of data that a computer can work with at any given time. Note, however, that adding extra CPU resources doesn't increase the speed at which data can be processed; any more than two cars can go twice as fast as one.

The second reason for breaking an application into multiple processes is that it makes it much easier for a developer to organize the elements of their solution. Consider a word processing application. At any given instant the word processor may be accepting text from the user, performing a spell check of the document, auto-saving the text into a file and sending pages to the printer. It is much easier to create each of these operations as an individual process rather than trying to write a single application that tries to perform all these actions at the same time.

The first personal computers were powered by a single CPU and gave the appearance of the ability to run multiple programs at the same time by rapidly switching between them. The advent of multi-CPU computers has now made it possible for systems to use genuine multi-tasking, where elements of a solution can run in parallel on the hardware.

Of course, with the great power provided by the ability to execute multiple processes comes an extra layer of responsibility. A developer must ensure that all the elements in a multi-process powered solution work together correctly and that any errors are propagated through the system in a meaningful way.

This chapter explains the fundamentals of multi-process programming and describes the C# features and libraries that you can use to work with processes, how processes are created, managed and communicate with each other. You will also explore the C# language features that allow the management of error conditions in multi-process systems and discover the fundamental C# elements that control program flow.

Skills in this chapter:

SKILL 1.1: IMPLEMENT MULTITHREADING AND ASYNCHRONOUS PROCESSING

Consider a busy kitchen that needs to serve a large number of diners at the same time. If that kitchen has only one chef, that chef must rapidly switch between cooking each meal. While one dish is in the oven baking, the chef must prepare the ingredients for another dish. At any given instant the chef can work on only one meal. This is an example of how a single processor (the chef) is *multi-tasking* between the preparation of several meals. When two things are happening at exactly the same time

You can improve the performance of your kitchen by adding extra chefs, each of which will be working on a particular meal at any one time. This is analogous to adding extra CPUs to a multi-tasking system. In a kitchen you might decide to allocate particular tasks to a particular chef, so all of the desserts can be prepared by the pastry chef. In a multi-threaded computer system this would not happen, because tasks are allocated to the next processor that becomes available. If one task is delayed for some reason, perhaps because it is waiting for some data to arrive from a mass storage device, then the processor running that task can move onto a different task.

This ability of a computer system to execute multiple processes at the same time (concurrency) is not provided by the C# language itself. It is the underlying operating system that controls which programs are active at any instant. The .NET framework provides classes to represent items of work to be performed, and in this section you learn how to use these classes.

It is not possible for a developer to make any assumptions concerning which processes are active at any one time, how much processing time a given process has, or when a given operation will be completed.

In this section you will also discover the abstractions used by C# to manage the simultaneous execution of program elements, how to add parallel elements to sequential programs, and how to manage access to data in applications that use concurrency.

This section covers how to:

- Use the Task Parallel library, including the `Parallel.For` method, PLINQ, and tasks
- Create continuation tasks
- Spawn threads by using ThreadPool
- Unblock the UI
- Use async and await keywords
- Manage data by using concurrent collections

The Task Parallel library

You can think of a task as an abstraction of a unit of work to be performed. The work itself will be described by some C# program code, perhaps by a method or a lambda expression. A task may be performed concurrently with other tasks.

The Task Parallel Library (TPL) provides a range of resources that allow you to use tasks in an application. The `Task.Parallel` class in the library provides three methods that can be used to create applications that contain tasks that execute in parallel.

`Parallel.Invoke`

The `Task.Parallel` class can be found in the `System.Threading.Tasks` namespace. The `Parallel.Invoke` method accepts a number of `Action` delegates and creates a `Task` for each of them.

An `Action` delegate is an encapsulation of a method that accepts no parameters and does not return a result. It can be replaced with a lamba expression, as shown in Listing 1-1, in which two tasks are created.

LISTING 1-1 `Parallel.Invoke` in use

Click here to view code image

```
using System;
using System.Threading.Tasks;
using System.Threading;

namespace Listing_1._1Parallel_Invoke
{
    class Program
    {
        static void Task1()
        {
            Console.WriteLine("Task 1 starting");
            Thread.Sleep(2000);
            Console.WriteLine("Task 1 ending");
        }

        static void Task2()
        {
            Console.WriteLine("Task 2 starting");
            Thread.Sleep(1000);
            Console.WriteLine("Task 2 ending");
        }

        static void Main(string[] args)
        {
            Parallel.Invoke(()=>Task1(), ()=>Task2(
            Console.WriteLine("Finished processing.
            Console.ReadKey();
        }
    }
}
```

The Parallel.Invoke method can start a large number of tasks at once. You have no control over the order in which the tasks are started or which processor they are assigned to. The Parallel.Invoke method returns when all of the tasks have completed. You can see the output from the program here.

Click here to view code image

```
Task 1 starting
Task 2 starting
Task 2 ending
Task 1 ending
Finished processing. Press a key to end.
```

Parallel.ForEach

The Task.Parallel class also provides a ForEach method that performs a parallel implementation of the foreach loop construction, as shown in Listing 1-2, in which the WorkOnItem method is called to process each of the items in a list.

LISTING 1-2 ParallelForEach in use

Click here to view code image

```
static void WorkOnItem(object item)
{
    Console.WriteLine("Started working on: " + item
    Thread.Sleep(100);
    Console.WriteLine("Finished working on: " + ite
}

static void Main(string[] args)
{
    var items = Enumerable.Range(0, 500);
    Parallel.ForEach(items, item =>
    {
        WorkOnItem(item);
    });

    Console.WriteLine("Finished processing. Press a
    Console.ReadKey();
}
```

The Parallel.ForEach method accepts two parameters. The first parameter is an IEnumerable collection (in this case the list items). The second parameter provides the action to be performed on each item in the list. You can see some of the output from this program below. Note that the tasks are not completed in the same order that they were started.

Click here to view code image

```
Finished working on: 472
Started working on: 473
Finished working on: 488
Started working on: 489
Finished working on: 457
Finished working on: 473
Finished working on: 489
Finished processing. Press a key to end.
```

Parallel.For

The Parallel.For method can be used to parallelize the execution of a for loop, which is governed by a control variable (see Listing 1-3).

LISTING 1-3 ParallelFor in use

Click here to view code image

```
static void Main(string[] args)
{
    var items = Enumerable.Range(0, 500).ToArray();

    Parallel.For(0, items.Length, i =>
    {
        WorkOnItem(items[i]);
    });
    Console.WriteLine("Finished processing. Press a
    Console.ReadKey();
}
```

This implements a counter starting at 0 (the first parameter of the Parallel.For method), for the length of the items array (the second parameter of the Parallel.For method). The third parameter of the method is a lambda expression, which is passed a variable that provides the counter value for each iteration. You can find out more about delegate and lambda expressions in the section, “Create and Implement Callbacks,” later in this chapter. The example produces the same output as Listing 1-2.

Managing Parallel.For and Parallel.Foreach

The lambda expression that executes each iteration of the loop can be

ParallelLoopResult that can be used to determine whether or not a parallel loop has successfully completed.

Listing 1-4 shows how these features are used. The code in the lambda expression checks the number of the work item (in the range 0 to 500). If the work item is number 200 the code calls the Stop method on the loopState value which is controlling this loop to request that the iterator stop running any more iterations. Note that this doesn't mean that the iterator will instantly stop any executing iterations. Note also that this doesn't mean that work items with a number greater than 200 will never run, because there is no guarantee that the work item with number 200 (which triggers the stop) will run before work items with higher numbers.

LISTING 1-4 Managing a parallel For loop

Click here to view code image

```
static void Main(string[] args)
{
    var items = Enumerable.Range(0, 500).ToArray();

    ParallelLoopResult result = Parallel.For(0, iter
    {
        if (i == 200)
            loopState.Stop();

        WorkOnItem(items[i]);
    });

    Console.WriteLine("Completed: " + result.IsComp
    Console.WriteLine("Items: " + result.LowestBrea

    Console.WriteLine("Finished processing. Press a
    Console.ReadKey();
}
```

The iterations can be ended by calling the Stop or Break methods on the ParallelLoopState variable. Calling Stop will prevent any new iterations with an index value greater than the current index. If Stop is used to stop the loop during the 200th iteration it might be that iterations with an index lower than 200 will not be performed. If Break is used to end the loop iteration, all the iterations with an index lower than 200 are guaranteed to be completed before the loop is ended.

Parallel LINQ

Language-Integrated Query, or LINQ, is used to perform queries on items of data in C# programs. *Parallel Language-Integrated Query* (PLINQ) can be used to allow elements of a query to execute in parallel. The code in Listing 1-5 creates a tiny dataset and then performs a parallel query on the data in it.

LISTING 1-5 A parallel LINQ query

Click here to view code image

```
using System;
using System.Linq;

namespace LISTING_1_5_A_parallel_LINQ_query
{
    class Program
    {
        class Person
        {
            public string Name { get; set; }
            public string City { get; set; }
        }

        static void Main(string[] args)
        {
            Person [] people = new Person [] {
                new Person { Name = "Alan", City = "
                new Person { Name = "Beryl", City =
                new Person { Name = "Charles", City
                new Person { Name = "David", City =
                new Person { Name = "Eddy", City = "
                new Person { Name = "Fred", City = "
                new Person { Name = "Gordon", City =
                new Person { Name = "Henry", City =
                new Person { Name = "Isaac", City =
                new Person { Name = "James", City =

            var result = from person in people.AsPa
                         where person.City == "Seat
                         select person;

            foreach (var person in result)
                Console.WriteLine(person.Name);

            Console.WriteLine("Finished processing.
            Console.ReadKey();
        }
    }
}
```

The AsParallel method examines the query to determine if using a parallel version would speed it up. If it is decided that executing elements of the query in parallel would improve performance, the query is broken down into a number of processes and each is run concurrently. If the AsParallel method can't decide whether parallelization would improve performance the query is not executed in parallel. If you really want to use AsParallel you should design the behavior with this in mind, otherwise performance may not be improved and it is possible that you might get the wrong outputs.

Informing parallelization

Programs can use other method calls to further inform the parallelization process, as shown in Listing 1-6.

LISTING 1-6 Informing parallelization

Click here to view code image

```
var result = from person in people.AsParallel().
                WithDegreeOfParallelism(4).
                WithExecutionMode(ParallelExecutionM
                where person.City == "Seattle"
                select person;
```

This call of AsParallel requests that the query be parallelized whether performance is improved or not, with the request that the query be executed on a maximum of four processors.

A non-parallel query produces output data that has the same order as the input data. A parallel query, however, may process data in a different order from the input data. In other words, the query in Listing 1-5 produces the following output.

Click here to view code image

```
Henry
Beryl
David
Issac
```

The name Henry is printed first, even though it is not the first item in the source data. If it is important that the order of the original data be preserved, the AsOrdered method can be used to request this from the query (see Listing 1-7).

LISTING 1-7 Using AsOrdered to preserve data ordering

Click here to view code image

```
var result = from person in
   people.AsParallel().AsOrdered()
                where person.City == "Seattle"
                select person;
```

The AsOrdered method doesn't prevent the parallelization of the query, instead it organizes the output so that it is in the same order as the original data. This can slow down the query.

Another issue that can arise is that the parallel nature of a query may remove ordering of a complex query. The AsSequential method can be used to identify parts of a query that must be sequentially executed (see Listing 1-8). AsSequential executes the query in order whereas AsOrdered returns a sorted result but does not necessarily run the query in order.

LISTING 1-8 Identifying elements of a parallel query as sequential

Click here to view code image

```
var result = (from person in people.AsParallel()
                    where person.City == "Seattle"
                    orderby (person.Name)
                    select new
                    {
                        Name = person.Name
                    }).AsSequential().Take(4) ;
```

The query in Listing 1-8 retrieves the names of the first four people who live in Seattle. The query requests that the result be ordered by person name, and this ordering is preserved by the use of AsSequential before the Take, which removes the four people. If the Take is executed in parallel it can disrupt the ordering of the result.

Iterating query elements using ForAll

The ForAll method can be used to iterate through all of the elements in a query. It differs from the foreach C# construction in that the iteration takes place in parallel and will start before the query is complete (see Listing 1-9).

LISTING 1-9 Using the ForAll method

Click here to view code image

```
                where person.City == "Seattle"
                select person;
result.ForAll(person => Console.WriteLine(person.Na
```

The parallel nature of the execution of `ForAll` means that the order of the printed output above will not reflect the ordering of the input data.

Exceptions in queries

It is possible that elements of a query may throw exceptions:

Click here to view code image

```
public static bool CheckCity(string name)
{
    if (name == "")
        throw new ArgumentException(name);
    return name == "Seattle";
}
```

This `CheckCity` method throws an exception when the city name is empty. Using this method in a PLINQ query (Listing 1-10) will cause exceptions to be thrown when empty city names are encountered in the data.

LISTING 1-10 Exceptions in PLINQ queries

Click here to view code image

```
try
{
    var result = from person in
        people.AsParallel()
                    where CheckCity(person.City)
                    select person;
    result.ForAll(person => Console.WriteLine(perso
}
catch (AggregateException e)
{
    Console.WriteLine(e.InnerExceptions.Count + " e
}
```

The code in Listing 1-10 uses the `CheckCity` method in a query. This will cause exceptions to be thrown when empty city names are encountered during the query. If any queries generate exceptions an `AgregateException` will be thrown when the query is complete. This contains a list, `InnnerExceptions`, of the exceptions that were thrown during the query.

Note that the outer catch of `AggregateException` does catch any exceptions thrown by the `CheckCity` method. If elements of a query can generate exceptions it is considered good programming practice to catch and deal with them as close to the source as possible.

Tasks

The parallelization tools covered in this chapter so far have operated at a very high level of abstraction. Tasks have been created, but the code hasn't interacted with them directly. Now let's consider how to create and manage tasks.

Create a task

The code in Listing 1-11 creates a task, starts it running, and then waits for the task to complete.

LISTING 1-11 Create a task

Click here to view code image

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace LISTING_1_11_Create_a_task
{
    class Program
    {
        public static void DoWork()
        {
            Console.WriteLine("Work starting");
            Thread.Sleep(2000);
            Console.WriteLine("Work finished");
        }

        static void Main(string[] args)
        {
            Task newTask = new Task(() => DoWork())
            newTask.Start();
            newTask.Wait();
        }
    }
}
```

LISTING 1-12 Run a task

Click here to view code image

```
static void Main(string[] args)
{
    Task newTask = Task.Run(() => DoWork());
    newTask.Wait();
}
```

Your application can use tasks in this way if you just want to start the tasks and have them run to completion.

Return a value from a task

A task can be created that will return a value, as shown in Listing 1-13, where the task returns an integer. Note that a program will wait for the task to deliver the result when the `Result` property of the `Task` instance is read.

LISTING 1-13 Task returning a value

Click here to view code image

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace LISTING_1_13_Task_returning_a_value
{
    class Program
    {
        public static int CalculateResult()
        {
            Console.WriteLine("Work starting");
            Thread.Sleep(2000);
            Console.WriteLine("Work finished");
            return 99;
        }
        static void Main(string[] args)
        {
            Task<int> task = Task.Run(() =>
            {
                return CalculateResult();
            });

            Console.WriteLine(task.Result);

            Console.WriteLine("Finished processing.
            Console.ReadKey();
        }
    }
}
```

The `Task.Run` method uses the `TaskFactory.StartNew` method to create and start the task, using the default task scheduler that uses the .NET framework thread pool. The `Task` class exposes a `Factory` property that refers to the default task scheduler.

You can create your own task scheduler or run a task scheduler in the synchronization context of another processor. You can also create your own `TaskFactory` if you want to create a number of tasks with the same configuration. The `Run` method, however, is the preferred way to create a simple task, particularly if you want to use the task with *async* and *await* (covered later in this chapter).

Wait for tasks to complete

The `Task.Waitall` method can be used to pause a program until a number of tasks have completed, as shown in Listing 1-14. This listing also illustrates an additional issue with the use of loop control variables when they are passed into lambda expressions. The loop counter is copied into a local variable called `taskNum` in the loop that creates each task. If the variable `i` was used directly in the lambda expression, all of the tasks would have number 10, which is the value of the limit of the loop.

LISTING 1-14 Task waitall

Click here to view code image

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace LISTING_1_14_Task_waitall
{
    class Program
    {
        public static void DoWork(int i)
        {
            Console.WriteLine("Task {0} starting",i
            Thread.Sleep(2000);
            Console.WriteLine("Task {0} finished",
        }

        static void Main(string[] args)
        {
            {
                int taskNum = i;  // make a local c
                                  // correct task n
                                    lambda expressi
                Tasks[i] = Task.Run( () => DoWork(t
            }
            Task.WaitAll(Tasks);

            Console.WriteLine("Finished processing.
            Console.ReadKey();
        }
    }
}
```

Another use for `Task.Waitall` is to provide a place where a program can catch any exceptions that may be thrown by tasks. Note that, as with exceptions generated by PLINQ queries, the exceptions are aggregated.

You can use `Task.WaitAny` to make a program pause until any one of a number of concurrent tasks completes. If you think of each task as a horse in a race; `WaitAll` will pause until all the horses have finished running, whereas `WaitAny` will pause until the first horse has finished running. In the same way that horses still run after the winner has finished, some tasks will continue to run after a `WaitAny` call has returned.

Continuation Tasks

A continuation task can be nominated to start when an existing task (the *antecedent* task) finishes. If the antecedent task produces a result, it can be supplied as an input to the continuation task. Continuation tasks can be used to create a "pipeline" of operations, with each successive stage starting when the preceding one ends.

Create a continuation task

Listing 1-15 shows how a continuation task can be created from a task. A Task object exposes a `ContinueWith` method that can be used to specify a continuation task.

The lambda expression that executes the continuation task is provided with a reference to the antecedent task, which it can use to determine if the antecedent completed successfully. You can add continuation tasks to tasks that deliver a result, in which case the continuation task can use the `Result` property of the antecedent task to obtain its input data.

LISTING 1-15 Continuation tasks

Click here to view code image

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace LISTING_1_15_Continuation_tasks
{
    class Program
    {
        public static void HelloTask()
        {
            Thread.Sleep(1000);
            Console.WriteLine("Hello");
        }

        public static void WorldTask()
        {
            Thread.Sleep(1000);
            Console.WriteLine("World");
        }

        static void Main(string[] args)
        {
            Task task = Task.Run(() => HelloTask())
            task.ContinueWith( (prevTask) => WorldT

            Console.WriteLine("Finished processing.
            Console.ReadKey();
        }
    }
}
```

The `ContinueWith` method has an overload that you can use to specify when a given continuation task can run. This version accepts a parameter of type `TaskContinuationOptions`. Listing 1-16 shows how these can be used.

LISTING 1-16 Continuation options

Click here to view code image

```
Task task = Task.Run(() => HelloTask());

task.ContinueWith((prevTask) => WorldTask(), TaskCo
                                OnlyOnRanToCompleti
task.ContinueWith((prevTask) => ExceptionTask(), Ta
```

The method `WorldTask` (the method to be performed by the continuation task) is now only called if the method `HelloTask` (the method run by the first task) completes successfully. If `HelloTask` throws an exception, a task will be started that runs the method `ExceptionTask`.

Child tasks

Code running inside a *parent* Task can create other tasks, but these "child" tasks will execute independently of the parent in which they were created. Such tasks are called *detached child tasks* or *detached nested tasks*. A parent task can create child tasks with a task creation option that specifies that the child task is attached to the parent. The parent class will *not* complete until all of the *attached child tasks* have completed.

Listing 1-17 shows a parent Task creating 10 attached child tasks. The tasks are created by calling the `StartNew` method on the default Task Factory provided by the Task class. This overload of the `StartNew` method accepts three parameters: the lambda expression giving the behavior of the task, a state object that is passed into the task when it is started, and a `TaskCreationOption` value that requests that the new task should be a child task.

LISTING 1-17 Attached child tasks

Click here to view code image

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace LISTING_1_17_Attached_child_tasks
{
    class Program
    {
        public static void DoChild(object state)
        {
            Console.WriteLine("Child {0} starting",
            Thread.Sleep(2000);
            Console.WriteLine("Child {0} finished",
        }

        static void Main(string[] args)
        {
            var parent = Task.Factory.StartNew(() =
                Console.WriteLine("Parent starts");
                for (int i = 0; i < 10; i++)
                {
                    int taskNo = i;
                    Task.Factory.StartNew(
                        (x) => DoChild(x), // lambda
                               taskNo, // state obj
                               TaskCreationOptions.
                }
            });

            parent.Wait(); // will wait for all the

            Console.WriteLine("Parent finished. Pre
            Console.ReadKey();
        }
    }
}
```

You can create a task without any attached child tasks by specifying the `TaskCreationOptions.DenyChildAttach` option when you create the task. Children of such a task will always be created as detached child tasks. Note that tasks created using the `Task.Run` method have the `TaskCreationOptions.DenyChildAttach` option set, and therefore can't have attached child tasks.

Threads and ThreadPool

Threads are a lower level of abstraction than tasks. A `Task` object represents an item of work to be performed, whereas a `Thread` object represents a process running within the operating system.

Threads and Tasks

When creating your first threads you will notice that the code looks rather similar to that used to create your first tasks, There are, however, some important differences between the two that you need to be aware of:

- Threads are created as *foreground* processes (although they can be set to run in the background). The operating system will run a foreground process to completion, which means that an application will not terminate while it contains an active foreground thread. A foreground process that contains an infinite loop will execute forever, or until it throws an uncaught exception or the operating system terminates it. Tasks are created as *background* processes. This means that tasks can be terminated before they complete if all the foreground threads in an application complete.

- Threads have a *priority* property that can be changed during the lifetime of the thread. It is not possible to set the priority of a task. This gives a thread a higher priority request so a greater portion of available processor time is allocated.

- A thread cannot deliver a result to another thread. Threads must

- It is not possible to create a continuation on a thread. Instead, threads provide a method called a join, which allows one thread to pause until another completes.

- It is not possible to aggregate exceptions over a number of threads. An exception thrown inside a thread must be caught and dealt with by the code in that thread. Tasks provide exception aggregation, but threads don't.

Create a thread

The `Thread` class is located in the `System.Threading` namespace. When you create a `Thread` you can pass the constructor the name of the method the thread will run. Once the thread has been created, you can call the `Start` method on the thread to start it running. Listing 1-18 shows how this is done.

LISTING 1-18 Creating threads

Click here to view code image

```
using System;
using System.Threading;

namespace LISTING_1_18_Creating_threads
{
    class Program
    {
        static void ThreadHello()
        {
            Console.WriteLine("Hello from the threa
            Thread.Sleep(2000);
        }

        static void Main(string[] args)
        {
            Thread thread = new Thread(ThreadHello)
            thread.Start();
        }
    }
}
```

Threads and ThreadStart

Note that earlier versions of .NET required the creation of a `ThreadStart` delegate to specify the method to be executed by the thread. Listing 1-19 shows how this is done. It's not currently necessary, but you may see it used in older programs.

LISTING 1-19 Using ThreadStart

Click here to view code image

```
static void Main(string[] args)
{
    ThreadStart ts = new ThreadStart(ThreadHello);
    Thread thread = new Thread(ts);
    thread.Start();
}
```

The `ThreadStart` delegate is no longer required.

Threads and lambda expressions

It is possible to start a thread using a lambda expression to specify the action of the thread, as shown in Listing 1-20.

LISTING 1-20 Threads and lambda expressions

Click here to view code image

```
static void Main(string[] args)
{
    Thread thread = new Thread(() =>
    {
        Console.WriteLine("Hello from the thread");
        Thread.Sleep(1000);
    });

    thread.Start();
    Console.WriteLine("Press a key to end.");
    Console.ReadKey();
}
```

When this program runs you might be surprised to see the output below:

Click here to view code image

```
Press a key to end.
Hello from the thread
```

It looks like the program is printing things in the wrong order. However, if you think about it, the ordering makes sense. The thread running inside to end." Then the background thread gets control and prints, "Hello from the thread."

Passing data into a thread

A program can pass data into a thread when it is created by using the `ParameterizedThreadStart` delegate. This specifies the thread method as one that accepts a single object parameter. The object to be passed into the thread is then placed in the `Start` method, as shown in Listing 1-21.

LISTING 1-21 ParameterizedThreadStart

Click here to view code image

```
using System;
using System.Threading;

namespace LISTING_1_21_ParameterizedThreadStart
{
    class Program
    {
        static void WorkOnData(object data)
        {
            Console.WriteLine("Working on: {0}", da
            Thread.Sleep(1000);
        }
        static void Main(string[] args)
        {
            ParameterizedThreadStart ps = new Param
            Thread thread = new Thread(ps);
            thread.Start(99);
        }
    }
}
```

Another way to pass data into a thread is to specify the behavior of the thread as a lambda expression that accepts a parameter. The parameter to the lambda expression is the data to be passed into the thread. Listing 1-22 shows how this is done; the parameter is given the name `data` in the lamba expression and the value `99` is passed into the lambda expression via the `Start` method.

LISTING 1-22 thread lambda parameters

Click here to view code image

```
static void Main(string[] args)
{
    Thread thread = new Thread((data) =>
    {
        WorkOnData(data);
    });
    thread.Start(99);
}
```

Note that the data to be passed into the thread is always passed as an object reference. This means that there is no way to be sure at compile time that thread initialization is being performed with a particular type of data.

Abort a thread

A `Thread` object exposes an `Abort` method, which can be called on the thread to abort it. The thread is terminated instantly. Listing 1-23 shows how one thread can abort another.

LISTING 1-23 Aborting a thread

Click here to view code image

```
using System;
using System.Threading;

namespace LISTING_1_23_aborting_a_thread
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread tickThread = new Thread(() =>
            {
                while (true)
                {
                    Console.WriteLine("Tick");
                    Thread.Sleep(1000);
                }
            });

            tickThread.Start();

            Console.WriteLine("Press a key to stop 
            Console.ReadKey();
            tickThread.Abort();
            Console.WriteLine("Press a key to exit"
            Console.ReadKey();
        }
```

When a thread is aborted it is instantly stopped. This might mean that it leaves the program in an ambiguous state, with files open and resources assigned. A better way to abort a thread is to use a shared flag variable. Listing 1-24 shows how to do this. The variable `tickRunning` is used to control the loop in `tickThread`. When `tickRunning` is set to `false` the thread ends.

LISTING 1-24 A shared flag variable

Click here to view code image

```
using System;
using System.Threading;

namespace LISTING_1_24_shared_flag_variable
{
    class Program
    {
        static bool tickRunning;  // flag variable

        static void Main(string[] args)
        {
            tickRunning = true;

            Thread tickThread = new Thread(() =>
            {
                while (tickRunning)
                {
                    Console.WriteLine("Tick");
                    Thread.Sleep(1000);
                }
            });

            tickThread.Start();

            Console.WriteLine("Press a key to stop
            Console.ReadKey();
            tickRunning = false;
            Console.WriteLine("Press a key to exit"
            Console.ReadKey();
        }
    }
}
```

Thread Synchronization using join

The `join` method allows two threads to synchronize. When a thread calls the `join` method on another thread, the caller of `join` is held until the other thread completes. Listing 1-25 shows how this works.

LISTING 1-25 Using join

Click here to view code image

```
Thread threadToWaitFor = new Thread(() =>
{
    Console.WriteLine("Thread starting");
    Thread.Sleep(2000);
    Console.WriteLine("Thread done");
});

threadToWaitFor.Start();
Console.WriteLine("Joining thread");
threadToWaitFor.Join();
Console.WriteLine("Press a key to exit");
Console.ReadKey();
```

Thread data storage and ThreadLocal

Listing 1-24 earlier, shows how threads can "share" variables, which are declared in the program that the thread is running within. In this program two threads made use of the same variable, `threadRunning`. One thread read from the variable, and another thread wrote into it.

If you want each thread to have its own copy of a particular variable, you can use the `ThreadStatic` attribute to specify that the given variable should be created for each thread. The best way to understand this is to consider that if the `threadRunning` variable in Listing 1-24 was made `ThreadStatic` it would not stop a thread if `threadRunning` was made `false`, because there would be a copy of `threadRunning` for each thread. Changes to the variable in one thread would not affect the value in another.

If your program needs to initialize the local data for each thread you can use the `ThreadLocal<t>` class. When an instance of `ThreadLocal` is created it is given a delegate to the code that will initialize attributes of threads. Listing 1-26 shows how you can use it. The `RandomGenerator` member of the class returns a random number generator that is to be used by a thread to produce random behaviors. You want each thread to have the same "random" behavior, so the `RandomGenerator` produces a new `Random` number generator with the same seed each time it is called.

LISTING 1-26 ThreadLocal

Click here to view code image

```
namespace LISTING_1_27_ThreadLocal
{
    class Program
    {
        public static ThreadLocal<Random> RandomGen
            new ThreadLocal<Random>(() =>
            {
                return new Random(2);
            });

        static void Main(string[] args)
        {
            Thread t1 = new Thread(() =>
           {
               for (int i = 0; i < 5; i++)
               {
                   Console.WriteLine("t1: {0}", Ran
                   Thread.Sleep(500);
               }
           });

            Thread t2 = new Thread(() =>
            {
                for (int i = 0; i < 5; i++)
                {
                    Console.WriteLine("t2: {0}", Ra
                    Thread.Sleep(500);
                }
            });
            t1.Start();
            t2.Start();
            Console.ReadKey();
        }
    }
}
```

When different threads use the value of their `RandomGenerator` they will all produce the same sequence of random numbers. This is the output from the program:

Click here to view code image

```
t2: 7
t1: 7
t2: 4
t1: 4
t1: 1
t2: 1
t2: 9
t1: 9
t1: 1
t2: 1
```

Thread execution context

A `Thread` instance exposes a range of context information, and some items can be read and others read and set. The information available includes the name of the thread (if any) priority of the thread, whether it is foreground or background, the threads *culture* (this contains culture specific information in a value of type *CultureInfo*) and the security context of the thread. The `Thread.CurentThread` property can be used by a thread to discover this information about itself. Listing 1-27 shows how the information can be displayed.

LISTING 1-27 Thread context

Click here to view code image

```
using System;
using System.Threading;

namespace LISTING_1_27_Thread_context
{
    class Program
    {
        static void DisplayThread(Thread t)
        {
            Console.WriteLine("Name: {0}", t.Name);
            Console.WriteLine("Culture: {0}", t.Cur
            Console.WriteLine("Priority: {0}", t.Pr
            Console.WriteLine("Context: {0}", t.Exe
            Console.WriteLine("IsBackground?: {0}",
            Console.WriteLine("IsPool?: {0}", t.IsT

        }
        static void Main(string[] args)
        {
            Thread.CurrentThread.Name = "Main metho
            DisplayThread(Thread.CurrentThread);

        }
    }
}
```

Threads, like everything else in C#, are managed as objects. If an application creates a large number of threads, each of these will require an object to be created and then destroyed when the thread completes. A *thread pool* stores a collection of reusable thread objects. Rather than creating a new `Thread` instance, an application can instead request that a process execute on a thread from the thread pool. When the thread completes, the thread is returned to the pool for use by another process. Listing 1-28 shows how this works. The `ThreadPool` provides a method `QueueUserWorkItem`, which allocates a thread to run the supplied item of work.

The item of work is supplied as a `WaitCallback` delegate. There are two versions of this delegate. The version used in Listing 1-28 accepts a state object that can be used to provide state information to the thread to be started. The other version of `WaitCallback` does not accept state information.

LISTING 1-28 Thread pool

Click here to view code image

```
using System;
using System.Threading;

namespace LISTING_1_28_Thread_pool
{
    class Program
    {
        static void DoWork(object state)
        {
            Console.WriteLine("Doing work: {0}", st
            Thread.Sleep(500);
            Console.WriteLine("Work finished: {0}",
        }

        static void Main(string[] args)
        {
            for (int i = 0; i < 50; i++)
            {
                int stateNumber = i;
                ThreadPool.QueueUserWorkItem(state
            }
            Console.ReadKey();
        }
    }
}
```

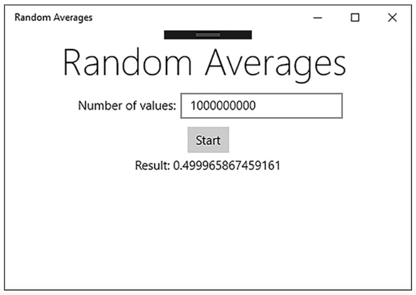
If you run the sample program in Listing 1-28 you discover that not all of the threads are started at the same time. The `ThreadPool` restricts the number of active threads and maintains a queue of threads waiting to execute. A program that creates a large number of individual threads can easily overwhelm a device. However, this does not happen if a `ThreadPool` is used. The extra threads are placed in the queue. Note that there are some situations when using the `ThreadPool` is not a good idea:

- If you create a large number of threads that may be idle for a very long time, this may block the ThreadPool, because the `ThreadPool` only contains a finite number of threads.
- You cannot manage the priority of threads in the `ThreadPool`.
- Threads in the `ThreadPool` have background priority. You cannot obtain a thread with foreground priority from the `ThreadPool`.
- Local state variables are not cleared when a ThreadPool thread is reused. They therefore should not be used.

Tasks and the User Interface

A Universal Application (Windows Store) Windows Presentation Foundation (WPF) application or a WinForms application can be regarded as having a single thread of execution that is dealing with the user interface. At any given instant this thread will be performing a particular action. In other words, when code in an event handler is running in response to a particular event, such as a button press, it is not possible for code in any other part of the user interface to execute. You sometimes see this behavior in badly written applications, where the user interface of the application becomes unresponsive while an action is carried out.

Figure 1-1 shows a simple Universal Windows application that calculates the average of a very large number of values.



The user can investigate the behavior of the C# random number generator over a very large number of operations. The number of averages to be generated is entered and when the Start button is pressed the program generates that number of random values and then prints out their average. Listing 1-29 shows the code for this application.

LISTING 1-29 Blocking the user interface

Click here to view code image

```
private double computeAverages(long noOfValues)
{
    double total = 0;
    Random rand = new Random();

    for (long values = 0; values < noOfValues; valu

    {
        total = total + rand.NextDouble();
    }

    return total / noOfValues;
}

private void StartButton_Click(object sender, Route
{
    long noOfValues = long.Parse(NumberOfValuesTextI
    ResultTextBlock.Text = "Result: " + computeAvera
}
```

Entering a very large number of averages causes the entire user interface to lock up while the program runs the event handler behind the "Start" button. The button appears to be "stuck down" for the time it takes the event handler to run and interactions with the user interface, for exampling resizing the application screen, are not possible until the button "pops back up" and the answer is displayed.

Tasks provide a means to solve this problem. Rather than performing an action directly, the event handler can instead start a task to perform the action in the background. The event handler then returns and the user interface can respond to other events. Listing 1-30 shows how a task can be used to perform the calculation in the background.

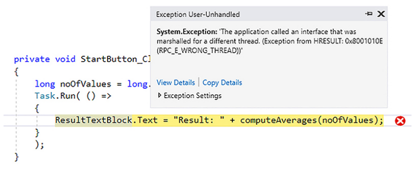
The idea behind this code is that the action of the button press starts a task running that completes in the background. This means that the button does not appear to "stick down" and lock up the user interface.

LISTING 1-30 Using a task

Click here to view code image

```
private void StartButton_Click(object sender, Route
{
    long noOfValues = long.Parse(NumberOfValuesTextI
    Task.Run( () =>
    {
        ResultTextBlock.Text = "Result: " + compute
    }
    );
}
```

This code is correct from a task management point of view, but it will fail when it runs. This is because interaction with display components is strictly managed by the process that generates the display. A background task cannot simply set the properties of a display element; instead it must follow a particular protocol to achieve the required update. Figure 1-2 shows what happens when the task attempts to display the result of the calculation.



```
private void StartButton_Cl
{
    long noOfValues = long.
    Task.Run( () =>
    {
        ResultTextBlock.Text = "Result: " + computeAverages(noOfValues);
    }
    );
}
```

Exception User-Unhandled

System.Exception: 'The application called an interface that was marshalled for a different thread. (Exception from HRESULT: 0x8001010E (RPC_E_WRONG_THREAD))'

FIGURE 1-2 Exception thrown by a display thread error

Listing 1-31 shows how the UI can be updated from a background task.

LISTING 1-31 Updating the UI

Click here to view code image

```
private void StartButton_Click(object sender, Route
{
    long noOfValues = long.Parse(NumberOfValuesTextI
    Task.Run(() =>
    {
        double result = computeAverages(noOfValues)

        ResultTextBlock.Dispatcher.RunAsync(CoreDis
        {
            ResultTextBlock.Text = "Result: " + resu
```

Each component on a display has a Dispatcher property that can be used to run tasks in the context of the display. The RunAsync method is given a priority level for the task, followed by the action that is to be performed on the thread. In the case of the code in Listing 1-31, the action to be performed is displaying the result.

Note that the code in Listing 1-31 does not show good programming practice. The RunAsync method is designed to be called asynchronously (you will discover what this means in the next section). The code in Listing 1-31 does not do this, which will result in compiler warnings being produced when you build the example. The best way to display a result from a task is to make use of async and await, as you will see in the next section.

Using async and await

A Task is a very useful way to get things done. It provides a means by which a program can partition and dispatch items of work. Tasks are particularly useful if a program has something else it can do while a given task is being performed. For example, a user interface can continue to respond to actions from the user while a long-running background task completes, or a web server can create tasks to assemble responses to web page requests. Tasks are particularly useful when performing actions that may take some time to complete, for example with input/output or network requests. If one task is waiting for a file to be loaded from a disk, another task can be assembling a message to be sent via the network.

A difficulty with tasks is that they can be hard for the programmer to manage. The application must contain code to create the Task and start it running, so the application must contain some means by which the code performing the Task can communicate that it has finished. If any of the actions that are being performed might generate exceptions these must be caught and passed back to the application.

The async and await keywords allow programmers to write code elements that execute asynchronously. The async keyword is used to flag a method as "asynchronous." An asynchronous method must contain one or more actions that are "awaited."

An action can be awaited if it returns either a Task (I just want do something asynchronously) or a Task<t> (I want to do something asynchronously that returns a result of a particular type).

The asyncComputeAverages method next returns a Task rather than a result. The task returns a double value, which is the computed average. You can regard the method as a "wrapper" around the original method, which creates a task that runs the method and delivers the result.

Click here to view code image

```
private Task<double> asyncComputeAverages(long noOf
{
    return Task<double>.Run(() =>
    {
        return computeAverages(noOfValues);
    });
}
```

The StartButton_Click method in Listing 1-32 below uses the asyncComputeAverages method to calculate the average value and then display the result in the ResultTextBlock. When the user presses the button, the event handler instantly returns. Then, after a short interval, the ResultTextBlock displays the average value.

LISTING 1-32 Using async

Click here to view code image

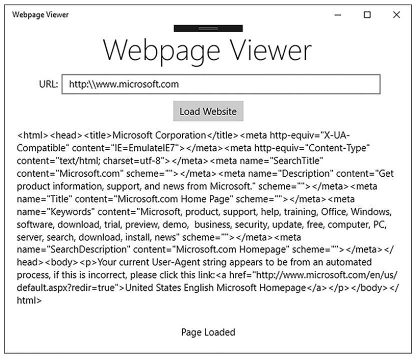
```
private async void StartButton_Click(object sender,
{
    long noOfValues = long.Parse(NumberOfValuesText

    ResultTextBlock.Text = "Calculating";

    double result = await (asyncComputeAverages(noO

    ResultTextBlock.Text = "Result: " + result.ToSt
}
```

So, how does this work? The StartButton_Click event handler method is marked as async. This tells the compiler to treat this method as special. It means that the method will contain one or more uses of the await keyword. The await keyword represents "a statement of intent" to perform an action. The keyword precedes a call of a method that will return the task to be performed. The compiler will generate code that will cause the async method to return to the caller at the point the await is reached. It will then go on to generate code that will perform the awaited action asynchronously and then continue with the body of the async method.

In the case of the Button_Click method in Listing 1-32, this means that the result is displayed upon completion of the task returned by

Exceptions and await/async

Figure 1-3 shows a simple Universal Windows application that can be used to view the text in a web page.



```
http:\\www.microsoft.com
```

```
<html><head><title>Microsoft Corporation</title><meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7"></meta><meta http-equiv="Content-Type" content="text/html; charset=utf-8"></meta><meta name="SearchTitle" content="Microsoft.com" scheme=""></meta><meta name="Description" content="Get product information, support, and news from Microsoft." scheme=""></meta><meta name="Title" content="Microsoft.com Home Page" scheme=""></meta><meta name="Keywords" content="Microsoft, product, support, help, training, Office, Windows, software, download, trial, preview, demo,  business, security, update, free, computer, PC, server, search, download, install, news" scheme=""></meta><meta name="SearchDescription" content="Microsoft.com Homepage" scheme=""></meta></head><body><p>Your current User-Agent string appears to be from an automated process, if this is incorrect, please click this link:<a href="http://www.microsoft.com/en/us/default.aspx?redir=true">United States English Microsoft Homepage</a></p></body></html>
```

Page Loaded

FIGURE 1-3 The Webpage Viewer application

The application uses an asynchronous method from the .NET library to fetch a web page from a given URL:

Click here to view code image

```
private async Task<string> FetchWebPage(string url)
{
    HttpClient httpClient = new HttpClient();
    return await httpClient.GetStringAsync(url);
}
```

The act of loading a web page may fail because the server is offline or the URL is incorrect. The `FetchWebPage` method will throw an exception in this situation. Listing 1-33 shows how this is used. The `await` is now enclosed in a try – catch construction. If an exception is thrown during the await, it can be caught and dealt with. In the case of the code in Listing 1-33, the `StatusTextBlock` is used to display the message from the exception.

LISTING 1-33 Exceptions and async

Click here to view code image

```
private async void Button_Click(object sender, Rout
{
    try
    {
        ResultTextBlock.Text = await FetchWebPage(UI
        StatusTextBlock.Text = "Page Loaded";
    }
    catch (Exception ex)
    {
        StatusTextBlock.Text = ex.Message;
    }
}
```

It is very important to note that exceptions can only be caught in this way because the `FetchWebPage` method returns a result; the text of the web page. It is possible to create an `async` method of type `void` that does not return a value. These are, however, to be avoided as there is no way of catching any exceptions that they generate. The only `async void` methods that a program should contain are the event handlers themselves, such as the `Button_Click` method in Listing 1-33. Even a method that just performs an action should return a status value so that exceptions can be caught and dealt with.

Awaiting parallel tasks

An `async` method can contain a number of awaited actions. These will be completed in sequence. In other words, if you want to create an "awaitable" task that returns when a number of parallel tasks have completed you can use the `Task.WhenAll` method to create a task that completes when a given lists of tasks have been completed. Listing 1-34 shows how this works. The task `FetchWebPages` returns uses the `FetchWebPage` method from Listing 1-32 to generate a list of strings containing the text from a given list of urls. The `Task.WhenAll` method is given a list of tasks and returns a collection which contains their results when they have completed.

LISTING 1-34 Awaiting parallel tasks

Click here to view code image

```
static async Task<IEnumerable<string>> FetchWebPage:
{
    var tasks = new List<Task<String>>();

    foreach (string url in urls)
    {
        tasks.Add(FetchWebPage(url));
        return await Task.WhenAll(tasks);
    }
```

Note that Listing 1-34 shows how `WhenAll` is used, but it doesn't necessarily show good programming practice. The order of the items in the returned collection may not match the order of the submitted site names and there is no aggregation of any exceptions thrown by the calls to `FetchWebPage`. There is also a `WhenAny` method that will return when any one of the given tasks completes. This works in the same way as the WaitAny method that you saw earlier.

Using concurrent collections

The phrase *thread safe* describes code elements that work correctly when used from multiple processes (tasks) at the same time. The standard .NET collections (including `List`, `Queue` and `Dictionary`) are not thread safe. The .NET libraries provide thread safe (concurrent) collection classes that you can use when creating multi-tasking applications:

Click here to view code image

```
BlockingCollection<T>
ConcurrentQueue<T>
ConcurrentStack<T>
ConcurrentBag<T>
ConcurrentDictionary<TKey, TValue>
```

BlockingCollection<T>

From a design perspective, it is best to view a task in a multi-threaded application as either a *producer* or a *consumer* of data. A task that both produces and consumes data is vulnerable to "deadly embrace" situations. If task A is waiting for something produced by task B, and task B is waiting for something produced by Task A, and neither task can run.

The `BlockingCollection<T>` class is designed to be used in situations where you have some tasks producing data and other tasks consuming data. It provides a thread safe means of adding and removing items to a data store. It is called a *blocking* collection because a `Take` action will block a task if there are no items to be taken. A developer can set an upper limit for the size of the collection. Attempts to `Add` items to a full collection are also blocked.

Listing 1-35 shows how a `BlockingCollection` is used. The program creates a thread that attempts to add 10 items to a `BlockingCollection`, which has been created to hold five items. After adding the 5th item this thread is blocked. The program also creates a thread that takes items out of the collection. As soon as the read thread starts running, and takes some items out of the collection, the writing thread can continue.

LISTING 1-35 Using BlockingCollection

Click here to view code image

```
using System;
using System.Collections.Concurrent;
using System.Threading.Tasks;

namespace LISTING_1_35_Using_BlockingCollection
{
    class Program
    {
        static void Main(string[] args)
        {
            // Blocking collection that can hold 5
            BlockingCollection<int> data = new Bloc

            Task.Run(() =>
            {
                // attempt to add 10 items to the c
                for(int i=0;i<11;i++)
                {
                    data.Add(i);
                    Console.WriteLine("Data {0} add
                }
                // indicate we have no more to add
                data.CompleteAdding();
            });

            Console.ReadKey();
            Console.WriteLine("Reading collection")

            Task.Run(() =>
            {
                while (!data.IsCompleted)
                {
                    try
                    {
                        int v = data.Take();
                        Console.WriteLine("Data {0}
                    }
                    catch (InvalidOperationExceptio
                }
            });

            Console.ReadKey();
```

The output from this program looks like this:

Click here to view code image

```
Data 0 added successfully.
Data 1 added successfully.
Data 2 added successfully.
Data 3 added successfully.
Data 4 added successfully.
 Reading collection
Data 0 taken successfully.
Data 1 taken successfully.
Data 2 taken successfully.
Data 3 taken successfully.
Data 4 taken successfully.
Data 5 taken successfully.
Data 5 added successfully.
Data 6 added successfully.
Data 7 added successfully.
Data 8 added successfully.
Data 9 added successfully.
Data 6 taken successfully.
Data 7 taken successfully.
Data 8 taken successfully.
Data 9 taken successfully.
```

The adding task calls the `CompleteAdding` on the collection when it has added the last item. This prevents any more items from being added to the collection. The task taking from the collection uses the `IsCompleted` property of the collection to determine when to stop taking items from it. The `IsCompleted` property returns true when the collection is empty and `CompleteAdding` has been called. Note that the `Take` operation is performed inside `try—catch` construction. The `Take` method can throw an exception if the following sequence occurs:

1. The taking task checks the `IsCompleted` flag and finds that it is false.
2. The adding task (which is running at the same time as the taking task) then calls the `CompleteAdding` method on the collection.
3. The taking task then tries to perform a `Take` from a collection which has been marked as complete.

Note that this does not indicate a problem with the way that the `BlockingCollection` works; it instead shows how you need to be careful when using any kind of data store from multiple tasks. The `BlockingCollection` class provides the methods `TryAdd` and `TryTake` that can be used to attempt an action. Each returns `true` if the action succeeded. They can be used with timeout values and cancellation tokens.

The `BlockingCollection` class can act as a wrapper around other concurrent collection classes, including `ConcurrentQueue`, `ConcurrentStack`, and `ConcurrentBag`. Listing 1-36 shows how this is done. The collection class to be used is given as the first parameter of the `BlockingCollection` constructor.

LISTING 1-36 Block ConcurrentStack

Click here to view code image

```
BlockingCollection<int> data = new BlockingCollecti
```

If you execute this example you will see that items are added and taken from the stack on a “last in—first out” basis. If you don’t provide a collection class the `BlockingCollection` class uses a `ConcurrentQueue`, which operates on a “first in-first out” basis. The `ConcurrentBag` class stores items in an unordered collection.

Concurrent Queue

The `ConcurrentQueue` class provides support for concurrent queues. The `Enqueue` method adds items into the queue and the `TryDequeue` method removes them. Note that while the `Enqueue` method is guaranteed to work (queues can be of infinite length) the `TryDequeue` method will return false if the dequeue fails. A third method, `TryPeek`, allows a program to inspect the element at the start of the queue without removing it. Note that even if the `TryPeek` method returns an item, a subsequent call of the `TryDequeue` method in the same task removing that item from the queue would fail if the item is removed by another task. Listing 1-37 shows how a concurrent queue is used. It places two strings on the queue, peeks the top of the queue, and then removes one item from it.

It’s possible for a task to enumerate a concurrent queue (a program can use the `foreach` construction to work through each item in the queue). At the start of the enumeration a concurrent queue will provide a snapshot of the queue contents.

LISTING 1-37 Concurrent queue

Click here to view code image

```
queue.Enqueue("Miles");
string str;
if (queue.TryPeek(out str))
    Console.WriteLine("Peek: {0}", str);
if (queue.TryDequeue(out str))
    Console.WriteLine("Dequeue: {0}", str);
```

When this program runs it prints out "Rob," because that is the item at the start of the queue; and a queue is a first in–first out data store.

Concurrent Stack

The `ConcurrentStack` class provides support for concurrent stacks. The `Push` method adds items onto the stack and the `TryPop` method removes them. There are also methods, `PushRange` and `TryPopRange`, which can be used to push or pop a number of items. Listing 1-38 shows how a `ConcurrentStack` is used.

LISTING 1-38 A concurrent stack

Click here to view code image

```
ConcurrentStack<string> stack = new ConcurrentStack
stack.Push("Rob");
stack.Push("Miles");
string str;
if (stack.TryPeek(out str))
    Console.WriteLine("Peek: {0}", str);
if (stack.TryPop(out str))
    Console.WriteLine("Pop: {0}", str);
Console.ReadKey();
```

When this program runs it prints out "Miles," because it is at the top of the stack, and the stack is a last in–first out data store.

ConcurrentBag

You can use a `ConcurrentBag` to store items when the order in which they are added or removed isn't important. The `Add` items puts things into the bag, and the `TryTake` method removes them. There is also a `TryPeek` method, but this is less useful in a `ConcurrentBag` because it is possible that a following TryTake method returns a different item from the bag. Listing 1-39 shows how a `ConcurrentBag` is used.

LISTING 1-39 A concurrent bag

Click here to view code image

```
ConcurrentBag<string> bag = new ConcurrentBag<strin
bag.Add("Rob");
bag.Add("Miles");
bag.Add("Hull");
string str;
if (bag.TryPeek(out str))
    Console.WriteLine("Peek: {0}", str);
if (bag.TryTake(out str))
    Console.WriteLine("Take: {0}", str);
```

When I ran this program in Listing 1-39 it printed the word "Hull," but there is no guarantee that it will do this when you run it, especially if multiple tasks are using the `ConcurrentBag`.

ConcurrentDictionary

A dictionary provides a data store indexed by a key. A `ConcurrentDictionary` can be used by multiple concurrent tasks. Actions on the dictionary are performed in an *atomic* manner. In other words, an update action on an item in the dictionary cannot be interrupted by an action from another task. A `ConcurrentDictionary` provides some additional methods that are required when a dictionary is shared between multiple tasks. Listing 1-40 shows how these are used.

LISTING 1-40 Concurrent dictionary

Click here to view code image

```
ConcurrentDictionary<string, int> ages = new Concur
if (ages.TryAdd("Rob", 21))
    Console.WriteLine("Rob added successfully.");
Console.WriteLine("Rob's age: {0}", ages["Rob"]);
// Set Rob's age to 22 if it is 21
if (ages.TryUpdate("Rob", 22, 21))
    Console.WriteLine("Age updated successfully");
Console.WriteLine("Rob's new age: {0}", ages["Rob"]
// Increment Rob's age atomically using factory met
Console.WriteLine("Rob's age updated to: {0}",
    ages.AddOrUpdate("Rob", 1, (name,age) => age =
Console.WriteLine("Rob's new age: {0}", ages["Rob"]
```

The `TryAdd` method tries to add a new item. If the item already exists, the `TryAdd` method returns `false`. The `TryUpdate` method is supplied with of the item indexed by “Rob” will only be updated to 22 if the existing value is 21. This allows a program to only update an item if it is at an expected value.

You might wonder when this is useful. Consider a situation where two processes decide that the age value of a person in the dictionary needs to be increased from 21 to 22. If both processes went ahead and increased the age value that would mean that the age would end up being 23, which is wrong. Instead, each process could use the `TryUpdate` method to try and increase the age. One process will succeed and change the age value to 22. The other process will fail, because it is not updating an age which is 21.

The `AddOrUpdate` method allows you to provide a behavior that will perform the update of a given item or add a new item if it does not already exist. In the case of the example above, the age of item “Rob” will not be set to 1 by the call of `AddOrUpdate`, because the item already exists. Instead, the action is performed, which increases the age of item “Rob” by 1. This action can be regarded as atomic, in that no other actions will be performed on this item until the update behavior has completed.

SKILL 1.2: MANAGE MULTITHREADING

You have seen how we can create threads and tasks and use concurrent collections to safely share data between them. Now it is time to dig a little deeper into tasks and processes and discover how to manage them.

When running some of the applications in the previous section we noticed that their behavior was not predictable. The order in which actions were performed was different each time a program ran. For example, the output from Listing 1-2 “ParallelForEach in use” is different each time you run it. This is because the operating system decides when a thread runs and the decisions are made based on the workload on the computer. If the machine suddenly gets very busy–perhaps performing an update–this will affect when threads get to run.

We call this way of working *asynchronous* because the operations are not synchronized. When you design asynchronous solutions that use multi-threading you must be very careful to ensure that uncertainty about the timings of thread activity does not affect the working of the application or the results that are produced. In this section you’ll discover how to synchronize access to resources that your application uses. You will see that if access to resources is not synchronized correctly it can result in programs calculating the wrong result. A badly written multi-threaded application might even get stuck because two processes are waiting for each other to complete. You’ll discover how to stop tasks that may have got stuck and how to ensure that threads work together irrespective of the order in which they are performed.

This section covers how to:

- Synchronize resources
- Implement locking
- Cancel a long-running task
- Implement thread-safe methods to handle race conditions

Resource synchronization

When an application is spread over several asynchronous tasks, it becomes impossible to predict the sequencing and timing of individual actions. You need to create applications with the understanding that any action may be interrupted in a way that has the potential to damage your application.

Let’s start with a simple application that adds up the numbers in an array. Listing 1-41 creates an array containing the values 0 to 500,000,000. It then uses a `for` loop to calculate the total of the array.

LISTING 1-41 Single task summing

Click here to view code image

```
using System;
using System.Linq;

namespace LISTING_1_41_Single_task_summing
{
    class Program
    {
        // make an array that holds the values 0 to
        static int[] items = Enumerable.Range(0, 50

        static void Main(string[] args)
        {
            long total = 0;

            for (int i = 0; i < items.Length; i++)
                total = total + items[i];

            Console.WriteLine("The total is: {0}",
            Console.ReadKey();
        }
    }
}
```

```
The total is: 1250000025000000
```

This is a single tasking solution that has to work through the entire array. You may decide to make use of the multiple processors in your computer and create a solution that creates a number of tasks, each of which will add up a particular area of the array.

Listing 1-42 serves as a refresher showing how tasks are created, and also illustrates how resource synchronization can cause problems in an application. It creates a number of tasks, each of which runs the method `addRangeOfValues`, which adds the contents of a particular range of values in the array to a total value. The idea is that the first task will add the values in the elements from 0 to 999, the second task will add the values in the elements 1000 to 1999, and so on up the array. The main method creates all of the tasks and then uses the `Task.WaitAll` method to cause the program to wait for the completion all of the tasks.

LISTING 1-42 Bad task interaction

Click here to view code image

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace LISTING_1_42_Bad_task_interaction
{

    class Program
    {
        static long sharedTotal;

        // make an array that holds the values 0 to
        static int[] items = Enumerable.Range(0, 50

        static void addRangeOfValues(int start, int
        {
            while (start < end)
            {
                sharedTotal = sharedTotal + items[s
                start++;
            }
        }

        static void Main(string[] args)
        {
            List<Task> tasks = new List<Task>();

            int rangeSize = 1000;
            int rangeStart = 0;

            while (rangeStart < items.Length)
            {
                int rangeEnd = rangeStart + rangeSi

                if (rangeEnd > items.Length)
                    rangeEnd = items.Length;

                // create local copies of the param
                int rs = rangeStart;
                int re = rangeEnd;

                tasks.Add(Task.Run(() => addRangeOf
                rangeStart = rangeEnd;
            }

            Task.WaitAll(tasks.ToArray());

            Console.WriteLine("The total is: {0}",
            Console.ReadKey();
        }
    }
}
```

You may expect that the program in Listing 1-42 prints out the same value as our original program. When running it on my machine, however, I received the following:

Click here to view code image

```
The total is: 98448836618
```

Apparently many updates to the variable `sharedTotal` didn't take place. What is happening here? The problem is caused by the way in which all of the tasks interact over the same shared value. Consider the following sequence of events:

1. Task number 1 starts performing an update of `sharedTotal`. It fetches the contents of the `sharedTotal` variable into the Central Processor Unit (CPU) and adds the contents of an array element to the value of `sharedTotal`. But, just as the CPU is about to write the result back into memory, the operating system stops task number 1 and switches to task number 2.

and then writes the result back into memory. Now the operating system returns control to task number 1.

3. Task number 1 writes the `sharedTotal` value it was working on from the CPU back into memory. This means that the update performed by task number 2 has been lost.

This is called a *race condition*. There is a race between two threads, and the behavior of the program is dependent on which threads first get to the `sharedTotal` variable. It's impossible to predict what a badly written program like this will do. I've seen situations where programs that contain this kind of mistake work perfectly on one kind of computer, and then fail when they run on a machine that has a smaller or larger number of processors. Remember that the nature of an *asynchronous* solution is that as programmers we really don't have any control over the order in which any parts of our system may execute.

Note that this threading issue can arise even if you use C# statements that look like they are atomic. The statement below adds 1 to the variable x. It looks like it is an atomic operation, but it actually involves reading, updating and storing the result, which are all steps that can be interrupted and may not be performed correctly owing to a race condition.

```
x += 1;
```

In the previous section you identified a need for *concurrent collections* that can be used by multiple asynchronous tasks. These collections have been implemented in a way that avoids problems like the one shown in Listing 1-42. Now let's investigate how to ensure that your own programs can work correctly in multi-threaded environments.

Implementing locking

When examining the `ConcurrentDictionary` collection we discovered that the actions on a dictionary that can be used by multiple processes are referred to as *atomic*. This means that an action by one process on a given dictionary entry cannot be interrupted by another process. The data corruption that you saw in Listing 1-41 is caused by the fact that adding one to a variable is not an atomic action. It can be interrupted, leading to tasks "fighting" over a single value.

Locks

A program can use *locking* to ensure that a given action is *atomic*. Atomic actions are performed to completion, so they cannot be interrupted. Access to an atomic action is controlled by a locking object, which you can think of as the keys to a restroom operated by a restaurant. To get access to the restroom you ask the cashier for the key. You can then go and use the restroom and, when finished, hand the key back to the cashier. If the restroom is in use when you request the key, you must wait until the person in front of you returns the key, so you can then go and use it.

Listing 1-43 shows how to create a locking object that works in the same way as the restroom key. The object is called `sharedTotalLock`, and it controls access to the statement that updates the value of `sharedTotal`. The `lock` statement is followed by a statement or block of code that is performed in an atomic manner, so it will not be possible for a task to interrupt the code protected by the lock. If you run the program now you will find that the correct value is printed out.

LISTING 1-43 Using locking

Click here to view code image

```
static object sharedTotalLock = new object();

static void addRangeOfValues(int start, int end)
{
    while (start < end)
    {
        lock (sharedTotalLock)
        {
            sharedTotal = sharedTotal + items[start
        }
        start++;
    }
}
```

At this point you may be pleased that you know how to create multi-threaded programs and use locks to prevent them from interacting in dangerous ways, but I have some bad news for you. While you have stopped the tasks from interacting in a dangerous manner, you've also removed any benefit from using multi-tasking. If you run the program in Listing 1-43 you will discover that it takes *longer* to sum the elements in the arrays than the previous versions.

This is because the tasks are not executing in parallel any more. Most of the time tasks are in a queue waiting for access to the shared total value. Adding a lock solved the problem of contention, but it has also stopped the tasks from executing in parallel, because they are waiting for access to a variable that they all need to use.

The solution to this problem is simple. Listing 1-44 shows a version of `addRangeOfValues`, which calculates a sub-total in the loop and works down the array adding up array elements. The sub-total is then added to the total value once this loop has completed. Rather than updating the shared total every time it adds a new element of the array; this version of the method only updates the shared total once. So there is now a

LISTING 1-44 Sensible locking

Click here to view code image

```
static void addRangeOfValues(int start, int end)
{
    long subTotal = 0;

    while (start < end)
    {
        subTotal = subTotal + items[start];
        start++;
    }
    lock (sharedTotalLock)
    {
        sharedTotal = sharedTotal + subTotal;
    }
}
```

When you create a parallel version of an operation you need to be mindful of potential value corruption when you use shared variables, and you should also carefully consider the impact of any locking that you use to prevent corruption. You also need to remember that when a task is running code protected by a lock, the task is in a position to block other tasks. This is similar to how a person taking a long time in the restroom will cause a long queue of people waiting to use it. Code in a lock should be as short as possible and should not contain any actions that might take a while to complete. As an example, your program should never perform input/output during a locked block of code.

Monitors

A `Monitor` provides a similar set of actions to a lock, but the code is arranged slightly differently. They allow a program to ensure that only one thread at a time can access a particular object. Rather than controlling a statement or block of code, as the `lock` keyword does, the atomic code is enclosed in calls of `Monitor.Enter` and `Monitor.Exit`. The Enter and Exit methods are passed a reference to an object that is used as the lock. In Listing 1-45 the lock object is `sharedTotalLock`. If you run the program you will find it behaves in exactly the same way as the one shown in Listing 1-43.

LISTING 1-45 Using monitors

Click here to view code image

```
static object sharedTotalLock = new object();

static void addRangeOfValues(int start, int end)
{
    long subTotal = 0;

    while (start < end)
    {
        subTotal = subTotal + items[start];
        start++;
    }

    Monitor.Enter(sharedTotalLock);
    sharedTotal = sharedTotal + subTotal;
    Monitor.Exit(sharedTotalLock);
}
```

If atomic code throws an exception, you need to be sure that any locks that have been claimed to enter the code are released. In statements managed by the lock keyword this happens automatically, if you use a `Monitor`, make sure that the lock is released.

In the case of the `Monitor` used in the `addRangeOfValues` method in Listing 1-45, there is no chance of the atomic code (the statement that updates the value of `sharedTotal`) throwing an exception. However, if an exception is thrown, it is important to ensure that `Monitor.Exit` is performed, otherwise this stops any other task from accessing the code.

Make sure that the `Monitor.Exit` method is always performed by enclosing the atomic code in a try block, and that you call `Monitor.Exit` in the final clause, which will always run:

Click here to view code image

```
Monitor.Enter(lockObject);
try
{
   // code that might throw an exception
}
finally
{
   Monitor.Exit(lockObject);
}
```

Note that if an atomic action throws an exception it indicates that something has gone wrong with your application. In such a situation, consider designing things so that the application reports the error and then terminates in the tidiest manner possible. It is possible that the

At this point you may be wondering why you might want to use monitors rather than locks. A program that uses the `lock` keyword has no way to check whether or not it will be blocked when it tries to enter the locked segment of code. If, however, a monitor is used, the program can do the following:

Click here to view code image

```
if (Monitor.TryEnter(lockObject))
{
    // code controlled by the lock
}
else
{
   // do something else because the lock object is 
}
```

The `TryEnter` method attempts to enter the code controlled by the lock. If this is not possible because the lock object is in use, the `TryEnter` method returns false. There are also versions of `TryEnter` that can atomically set a flag variable to indicate whether or not the lock was obtained, along with variable that will wait for the lock for a given number of milliseconds before giving up. These features add extra flexibility to task design.

Deadlocks in multi-threaded code

When looking at the use of the `BlockingCollection` in "Implement multithreading" we considered the problem posed by a *deadly embrace,* where two different tasks are waiting for each other to perform an action on a shared collection, which blocks from adding items when the collection is full and removing items when the collection is empty. This situation is also called *a deadlock*. The application in Listing 1-46 contains two methods and two lock objects. The methods use the lock objects in different order so that each task gets a lock object each, and then waits for the other lock object to become free. When the program runs, the two methods are called one after another.

LISTING 1-46 Sequential locking

Click here to view code image

```
using System;

namespace LISTING_1_46_Sequential_locking
{

    class Program
    {
        static object lock1 = new object();
        static object lock2 = new object();

        static void Method1()
        {
            lock (lock1)
            {
                Console.WriteLine("Method 1 got loc
                Console.WriteLine("Method 1 waiting
                lock (lock2)
                {
                    Console.WriteLine("Method 1 got
                }
                Console.WriteLine("Method 1 release
            }
            Console.WriteLine("Method 1 released lo
        }

        static void Method2()
        {
            lock (lock2)
            {
                Console.WriteLine("Method 2 got loc
                Console.WriteLine("Method 2 waiting
                lock (lock1)
                {
                    Console.WriteLine("Method 2 got
                }
                Console.WriteLine("Method 2 release
            }
            Console.WriteLine("Method 2 released lo
        }

        static void Main(string[] args)
        {
            Method1();
            Method2();
            Console.WriteLine("Methods complete. Pr
            Console.ReadKey();
        }
    }
}
```

Running this program creates the following output:

Click here to view code image

```
Method 1 got lock 2
Method 1 released lock 2
Method 1 released lock 1
Method 2 got lock 2
Method 2 waiting for lock 1
Method 2 got lock 1
Method 2 released lock 1
Method 2 released lock 2
Methods complete. Press any key to exit.
```

The program runs to completion. Each method gets the lock objects in turn because they are running sequentially. You can change the program so that the methods are performed by tasks. Listing 1-47 shows a main method that creates two tasks that will run the methods concurrently.

LISTING 1-47 Deadlocked tasks

Click here to view code image

```
static void Main(string[] args)
{
    Task t1 = Task.Run(() => Method1());
    Task t2 = Task.Run(() => Method2());
    Console.WriteLine("waiting for Task 2");
    t2.Wait();
    Console.WriteLine("Tasks complete. Press any ke
    Console.ReadKey();
}
```

Running this program generates the following output:

Click here to view code image

```
waiting for Task 2
Method 1 got lock 1
Method 1 waiting for lock 2
Method 2 got lock 2
Method 2 waiting for lock 1
```

The tasks in this case never complete. Each task is waiting for the other's lock object, and neither can continue. Note that this is not the same as creating an *infinite loop*, in which a program repeats a sequence of statements forever. You will not find that the program error in Listing 1-47 will use up the entire CPU in the same way as an infinitely repeated loop will do. Instead, these two tasks will sit in memory unable to do anything.

Writing the example in Listing 1-47 was difficult for me to do. When using synchronization objects, I make sure to never "nest" one use of a lock inside another.

The lock object

An incorrect use of the `lock` statement can introduce deadlocks into your applications. Any object managed by reference can be used as a locking object, which represents "the key to our restroom." The scope of the object should be restricted to the part of your application containing the cooperating tasks; remember that access to a lock object provides a means by which other code can lock out your tasks.

It is important to carefully consider the object that is to be used as a lock. It may be tempting to use a data object, or the reference to "this" in a method as the lock, but this is confusing. I recommend a policy of explicitly creating an object to be used as a lock.

It is not a good idea to use a string as a locking object, because the .NET string implementation uses a pool of strings during compilation. Every time a program assigns text to a string, the pool is checked to see if a string contains that text already. If the text is already in the string pool, the program uses a reference to it. In other words, the string "the" would only be stored once when an application is running. If two tasks use the same word as their locking object, they are sharing that lock, which leads to problems.

Interlocked operations

You saw that when we using multiple tasks to sum the contents of an array, you must protect access to the shared total by means of a lock. There's a better way of achieving thread safe access to the contents of a variable, which is to use the `Interlocked` class. This provides a set of thread-safe operations that can be performed on a variable. These include increment, decrement, exchange (swap a variable with another), and add.

Listing 1-48 shows how a program can use an interlocked version of `Add` to update the shared total with a value calculated by the `addRangeOfValues` in the array summing program.

LISTING 1-48 Interlock total

Click here to view code image

```
static void addRangeOfValues(int start, int end)
{
    long subTotal = 0;

    while (start < end)
    {
        Interlocked.Add(ref sharedTotal, subTotal);
    }
```

There is also a compare and exchange method that can be used to create a multi-tasking program to search through an array and find the largest value in that array.

Volatile variables

The source of a C# program goes through a number of stages before it is actually executed. The compilation process includes the examination of the sequence of statements to discover ways that a program can be made to run more quickly. This might result in statements being executed in a different order to the order they were written. Consider the following sequence of statements.

Click here to view code image

```
int x;
int y=0;
x = 99;
y = y + 1;
Console.WriteLine("The answer is: {0}", x);
```

The first statement assigns a value to variable x, the second does some work on the variable y, and the third statement prints the value in variable x. After compilation we may find that the order of the first two statements has been swapped, so that the value in x can be held inside the computer processor rather than having to be re-loaded from memory for the write statement.

In a single threaded situation this is perfectly acceptable, but if multiple threads are working on the code, this may result in unexpected behaviors. Furthermore, if another task changes the value of x while statements are running, and if the C# compiler caches the value of x between statements, this results in an out of date value being printed. C# provides the keyword `volatile`, which can be used to indicate that operations involving a particular variable are not optimized in this way.

Click here to view code image

```
volatile int x;
```

Operations involving the variable `x` will now not be optimized, and the value of `x` will be fetched from the copy in memory, rather than being cached in the processor. This can make operations involving the variable `x` a lot less efficient.

Cancelling a long-running task

When covering threads in the previous section we noted that it was possible for a thread to be aborted using the `Abort` method that can be called on an active `Thread` instance. In this section we'll investigate how an application can stop an executing `Task`.

The Cancellation Token

There is an important difference between threads and tasks, in that a `Thread` can be aborted at any time, whereas a `Task` must monitor a *cancellation token* so that it will end when told to.

Listing 1-49 shows how a task is cancelled. The `Clock` method is run as a task and displays a "tick" message every half second. The loop inside the `Clock` method is controlled by an instance of the `CancellationTokenSource` class. This instance is shared between the `Task` running the clock and the foreground program. The `CancellationTokenSource` instance exposes a property called `IsCancellationRequested`. When this property becomes `true` the loop in the `Clock` method completes and the task ends. This means that a `Task` has the opportunity to tidy up and release resources when it is told that it is being cancelled.

LISTING 1-49 Cancel a task

Click here to view code image

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace LISTING_1_49_cancel_a_task
{
    class Program
    {
        static CancellationTokenSource cancellation
        new CancellationTokenSource();

        static void Clock()
        {
            while (!cancellationTokenSource.IsCance
            {
                Console.WriteLine("Tick");
                Thread.Sleep(500);
            }
        }
```

```
                Task.Run(() => Clock());
                Console.WriteLine("Press any key to sto
                Console.ReadKey();
                cancellationTokenSource.Cancel();
                Console.WriteLine("Clock stopped");
                Console.ReadKey();
            }
        }
    }
```

If you run the program in Listing 1-49 you will find that the clock will continue ticking until you press a key and trigger the call of the `Cancel` method on the `cancellationTokenSource` object.

Raising an exception when a task is cancelled

A `Task` can indicate that it has been cancelled by raising an exception. This can be useful if a task is started in one place and monitored in another. The `Clock` method below ticks 20 times and then exits. It is supplied with a `CancellationToken` reference as a parameter, which is tested each time around the tick loop. If the task is cancelled, it throws an exception.

Click here to view code image

```
    static void Clock(CancellationToken cancellationTok
    {
        int tickCount = 0;

        while (!cancellationToken.IsCancellationRequest
        {
            tickCount++;
            Console.WriteLine("Tick");
            Thread.Sleep(500);
        }

        cancellationToken.ThrowIfCancellationRequested(
    }
```

Listing 1-50 shows how this method is used in a task. A `CancellationTokenSource` instance is created and the `token` from this instance is passed into the task running the `Clock` method. The clock method uses the variable `tickCount` to count the number of times it has gone around the tick loop. When `tickCount` reaches 20, the `Clock` method completes. A task instance exposes an `IsCompleted` property that indicates whether or not a task completes correctly. This property is tested when the user presses a key during a run of the program. If a key is pressed before the clock task is complete, the clock task is cancelled. The `ThrowIfCancellationRequested` method is called in the `Clock` method to throw an exception if the task has been cancelled. This exception is captured and displayed in the main method.

LISTING 1-50 Cancel with exception

Click here to view code image

```
            static void Main(string[] args)
            {
                CancellationTokenSource cancellationTok
                new CancellationTokenSource();

                Task clock = Task.Run(() => Clock(cance

                Console.WriteLine("Press any key to sto

                Console.ReadKey();

                if (clock.IsCompleted)
                {
                    Console.WriteLine("Clock task compl
                }
                else
                {
                    try
                    {
                        cancellationTokenSource.Cancel(
                        clock.Wait();
                    }
                    catch (AggregateException ex)
                    {
                        Console.WriteLine("Clock stoppe
                             ex.InnerExceptions[0].ToSt
                    }
                }
                Console.ReadKey();
            }
```

This code requires careful study. Note that if you run the program in Visual Studio and press a key to interrupt the clock, Visual Studio will report an unhandled exception in the `Clock` method. If you let the program continue you will find that it will then reach the exception handler in the `Main` method. This happens because Visual Studio is detected and only the exception handler for the AggregateException would run.

Implementing thread-safe methods

An object can provide services to other objects by exposing methods for them to use. If an object is going to be used in a multi-threaded application it is important that the method behaves in a *thread safe* manner. Thread safe means that the method can be called from multiple tasks simultaneously without producing incorrect results, and without placing the object that it is a member of into an invalid state.

Thread safety and member variables

You have seen the difficulties that can be caused by race conditions appearing when two tasks use a shared variable. Without locking, the program that attempts to use multiple tasks to sum the contents of an array fails because there is unmanaged access to the shared total value. You can see the same issues arising when a method uses the value of a member of an object.

Consider the class Counter in Listing 1-51. It is intended to be used to collect values and add them to a total value that can then be read by users of the object. It works well in a single threaded application. In an application that uses multiple tasks, however, it fails in the same way as the program shown in Listing 1-42. If several tasks make use of the IncreaseCounter method at the same time, race conditions cause updates to totalValue to be overwritten.

LISTING 1-51 Unsafe thread method

Click here to view code image

```
class Counter
{
    private int totalValue = 0;

    public void IncreaseCounter(int amount)
    {
        totalValue = totalValue + amount;
    }

    public int Total
    {
        get { return totalValue; }
    }
}
```

You can get the program to calculate the correct result by using an interlocking operation to update the totalValue member as shown in Listing 1-42. The problem is that you need to know that you have to do this. You've seen how easy it is to add multi-tasking to an existing application but be sure that all of the objects used in the application contain thread safe code. Otherwise, the application may be prone to the worst kind of errors; those that appear sporadically and inconsistently across different platforms.

Any use of a member of a class must be thread safe, and this must be done in a way that does not compromise multi-threaded performance. You've seen how adding locks can make things thread safe, but you also saw how doing this incorrectly can actually make performance worse. This may mean that creating a multi-tasked implementation of a system involves a complete re-write, with processes refactored as either producers or consumers of data.

Thread safety and method parameters

Parameters passed into a method by value are delivered as copies of the data that was given in the arguments to the method call. They are unique to that method call and cannot be changed by code running in any other task. Objects on the end of reference parameters, however, are susceptible to change by code running in other tasks. As an illustration of what can go wrong, consider a factory method called CreateCustomerFromRawData:

Click here to view code image

```
Customer CreateCustomerFromRawData( RawData inputIn
{
    // validates the information in inputInfo and c
    // if the information is valid
}
```

The CreateCustomerFromRawData method validates the input information held in a RawData instance and, if the information is correct, it creates a new Customer built from this data and returns it. If, however, RawData is a reference to an object, there is nothing to stop another task from changing the contents of this object at the same time as the CreateCustomer method is running. This can lead to a Customer instance being created that contains invalid or inconsistent data and generates errors that are hard to trace.

There are two ways that this issue can be addressed. The first is to make the RawData object a struct type, which will be passed by value into the method call. The second method is to create an atomic action that copies the incoming data into local variables that are specific to that call of the method. Either way, it is important to consider the ramifications of parallel execution when creating methods that accept reference

SKILL 1.3 IMPLEMENT PROGRAM FLOW

Now that you have spent some time considering how to create and manage threads we can take a look at what a program actually does. A program is a sequence of instructions that is followed that works on some input and produces an output. You can think of the execution process as flowing through the C# statements in the same way that water might flow down a riverbed. In this section we will consider the elements of C# that are associated with program flow.

Programs frequently have to work with large collections of data, and C# provides a range of collection types. From a program flow point of view, a program frequently has to work through such collections and you will explore the C# constructions that you can use to do this.

You will also discover the C# constructions that allow a program to react to the values in the data that it is working with and change its behavior appropriately. You will explore the `if` constructions that allows a program to make a decision. You will also take a look at the logical expressions that are evaluated to control and if condition and the switch construction that makes it easier to create decisions

And finally, a program must perform the actual data processing element by evaluating the results of expressions and using the values to update the contents of variables. You'll take a look at how expressions are evaluated in C# programs.

This section covers how to:

- Iterate across collection and array items
- Program decisions by using switch statements, if/then, and operators
- Evaluate expressions

Iterating across collections

C# provides a number of constructions that can implement looping behaviors: the *while* loop, the *do* loop, and the *for* loop. Let's examine each of these in turn.

The while loop construction

A while loop construction will perform a given statement or block, while a given logical expression has a true value. The code in Listing 1-52 is an example of an *infinite loop* that will never terminate.

LISTING 1-52 While loops

Click here to view code image

```
while(true)
{
    Console.WriteLine("Hello");
}
```

This is completely legal C# code that will write Hello a great many times. In contrast, the code here will never write Hello:

Click here to view code image

```
while(false)
{
    Console.WriteLine("Hello");
}
```

This illustrates an important aspect of the `while` loop. The condition that controls the looping behavior is tested before the statements controlled by the loop are obeyed. The `while(false)` loop will produce a compiler warning because the compiler will detect that the loop contains statements that are unreachable. A `while` construction can be used with a counter to repeat an action a number of times, as shown here:

Click here to view code image

```
int count = 0;
while(count < 10)
{
    Console.WriteLine("Hello {0}", count);
    count = count + 1;
}
```

This will print the Hello message 10 times. If you wish to use a counter in this manner you should instead make use of a `for` loop, which is described next. A `while` loop is very effective when creating a consumer of data. The construction next consumes data while it is available.

Click here to view code image

```
while(dataAvailable())
{
    processData();
}
```

The do-while loop construction also uses a logical expression to control the execution of a given statement or block. In the case of this loop, however, the condition is tested after the block has been performed once. The code in Listing 1-53 shows a do-while construction.

LISTING 1-53 Do–while loops

Click here to view code image

```
do
{
    Console.WriteLine("Hello");
} while (false);
```

The important thing to note about this code is that although the logical expression controlling it is false, which means that the loop will never repeat, the message Hello will be printed once, since the printing takes place *before* the logical expression is tested.

A do-while construction is useful when you want to create code that continuously fetches data until a valid value is entered:

Click here to view code image

```
do
{
   requestData();
while (!dataValid());
```

The for loop construction

A loop that is not infinite (one that should terminate at some point) can be made up of three things:

1. Initialization that is performed to set the loop up
2. A test that will determine if the loop should continue
3. An update to be performed each time the action of the loop has been performed

The for loop provides a way of creating these three elements in a single construction. Listing 1-54 shows how this works. Each of the actions made into a method that is called by the loop construction.

LISTING 1-54 For loops

Click here to view code image

```
using System;

namespace LISTING_1_54_for_loops
{
    class Program
    {
        static int counter;

        static void Initalize()
        {
            Console.WriteLine("Initialize called");
            counter = 0;
        }

        static void Update()
        {
            Console.WriteLine("Update called");
            counter = counter + 1;
        }

        static bool Test()
        {
            Console.WriteLine("Test called");
            return counter < 5;
        }
        static void Main(string[] args)
        {
            for(Initalize(); Test();  Update() )
            {
                Console.WriteLine("Hello {0}", coun
            }
            Console.ReadKey();
        }
    }
}
```

When you run this program, the following output is displayed:

Click here to view code image

```
Initialize called
Test called
Hello 0
Update called
Test called
Hello 1
Update called
```

```
Test called
Hello 3
Update called
Test called
Hello 4
Update called
Test called
```

This output shows that a test is performed immediately after initialization, so it is possible that the statement controlled by the loop may never be performed. This illustrates a very important aspect of the `for` loop construction, in that the initialize, test, and update behaviors can be anything that you wish.

A more conventional use of the `for` loop construction to repeat an action five times is shown below. A variable local to the code to be repeated is created within the `for` loop. This variable, called `counter`, is then printed by the code in the loop.

Click here to view code image

```
for(int counter = 0; counter < 5; counter++)
{
    Console.WriteLine("Hello {0}", counter);
}
```

You can leave out any of the elements of a `for` loop. You can also perform multiple statements for the initialize, update, and test elements. The statements are separated by a comma. Note that while it is possible to do exotic things like this, I advise that you don't do this in your programs.

Programming is not a place to show how clever you are, because it is more important that it is a place where you create code that is easy to understand. It is very unlikely that your "clever" construction will be more efficient than a much simpler one. And even if your clever construction is a bit faster, person time (the time spent by someone trying to understand your "clever" code) is much more expensive than computer time.

The foreach construction

It is perfectly possible to use a `for` loop to iterate through a collection of items. Listing 1-55 shows how a `for` loop can create an index variable used to obtain successive elements in an array of names.

LISTING 1-55 Iterate with for

Click here to view code image

```
using System;

namespace LISTING_1_55_iterate_with_for
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] names = { "Rob", "Mary", "Davi

            for (int index = 0; index < names.Lengt
            {
                Console.WriteLine(names[index]);
            }
            Console.ReadKey();
        }
    }
}
```

The `foreach` construction makes iterating through a collection much easier. Listing 1-56 shows how a `foreach` construction is used. Each time around the loop, the value of `name` is loaded with the next name in the collection.

LISTING 1-56 Iterate with foreach

Click here to view code image

```
foreach(string name in names)
{
    Console.WriteLine(name);
}
```

Note how the type of the iterating value must match the type of the items in the collection. In other words, the following code generates a compilation error, because the names array holds a collection of strings, not integers.

Click here to view code image

```
foreach(int name in names)
{
    Console.WriteLine(name);
}
```

It isn't possible for code in a `foreach` construction to modify the

Click here to view code image

```
foreach(string name in names)
{
    name = name.ToUpper();
}
```

If the `foreach` loop is working on a list of references to objects, the objects on the ends of those references can be changed. The code in Listing 1-57 works through a list of `Person` objects, changing the `Name` property of each person in the list to upper case. This compiles and runs correctly.

LISTING 1-57 Uppercase Person

Click here to view code image

```
foreach(Person person in people)
{
    person.Name = person.Name.ToUpper();
}
```

The `foreach` construction can iterate through any object which implements the `IEnumerable` interface. These objects expose a method called `GetIterator()`. This method must return an object that implements the `System.Collections.IEnumerator` interface. This interface exposes methods that the `foreach` construction can use to get the next item from the enumerator and determine if there any more items in the collection. Many collection classes, including lists and dictionaries, implement the `IEnumerable` interface. In chapter 2, "Create and implement a class hierarchy," you will discover how to make a class that implements the `IEnumerable` interface.

Note that the iteration can be implemented in a "lazy" way; the next item to be iterated only needs to be fetched when requested. The results of database queries can be returned as objects that implement the `IEnumerable` interface and then only fetch the actual data items when needed. It is important that the item being iterated is not changed during iteration, if the iterating code tried to remove items from the list it was iterating through this would cause the program to throw an exception when it ran.

The break statement

Any of the above loop constructions can be ended early by the use of a `break` statement. When the `break` statement is reached, the program immediately exits the loop. Listing 1-58 shows how `break` works. The loop is ended when it reaches the name "David."

LISTING 1-58 Using break

Click here to view code image

```
for (int index = 0; index < names.Length; index++)
{
    Console.WriteLine(names[index]);
    if (names[index] == "David")
        break;
}
```

A loop can many `break` statements, but from a design point of view this is to be discouraged because it can make it much harder to discern the flow through the program.

The continue statement

The `continue` statement does not cause a loop to end. Instead, it ends the current pass through the code controlled by the loop. The terminating condition is then tested to determine if the loop should continue. Listing 1-59 shows how continue is used. The program will not print out the name "David" because the conditional statement will trigger, causing the continue statement to be performed, and abandoning that pass through the loop.

LISTING 1-59 Using continue

Click here to view code image

```
for (int index = 0; index < names.Length; index++)
{
    if (names[index] == "David")
        continue;

    Console.WriteLine(names[index]);
}
```

Program decisions

As a program runs, it can make decisions. There are two program constructions that can be used to conditionally execute code: the if construction and the `switch` construction.

The if construction

the logical expression is `true` the statement is obeyed. An if construction can have an else element that contains code that is to be executed when the Boolean expression evaluates to `false`. The code in Listing 1-60 shows how this works.

LISTING 1-60 If construction

Click here to view code image

```
if(true)
{
    Console.WriteLine("This statement is always per
}
else
{
    Console.WriteLine("This statement is never perf
}
```

The `else` element of an `if` construction is optional. It is possible to "nest" if constructions inside one another.

Click here to view code image

```
if (true)
{
    Console.WriteLine("This statement is always per
    if (true)
    {
        Console.WriteLine("This statement is always
    }
    else
    {
        Console.WriteLine("This statement is never p
    }
}
```

Not all `if` constructions are required to have `else` elements. There is never any confusion about which `if` condition a given `else` binds to, since it is always the "nearest" one. If you want to modify this binding you can use braces to force different bindings, as you can see here. Note that the indenting also helps to show the reader the code statements that are controlled by each part of the `if` construction.

Click here to view code image

```
if(true)
{
    Console.WriteLine("This statement is always per
    if (true)
    {
        Console.WriteLine("This statement is always
    }
}
else
{
    Console.WriteLine("This statement is never perf
}
```

Logical expressions

A logical expression evaluates to a logical value, which is either true or false. We've seen that true and false are Boolean literal values and can be used in conditions, although this is not terribly useful. A logical expression can contain operators used to compare values. Relational operators are used to compare two values (see Table 1-1).

TABLE 1-1 Relational operators

Name	Operator	Behavior
Less than	<	True if the left-hand operand is less than the right-hand operand
Greater than	>	True if the left-hand operand is greater than the right-hand operand
Less than or equals	<=	True if the left-hand operand is less than or equal to the right-hand operand
Greater than or equals	<=	True if the left-hand operand is greater than or equal to the right-hand operand

Relational operators can be used between numeric variables and strings. In the context of a string, less than and greater than are evaluated alphabetically, as in "Abbie" is less than "Allen." A program can use equality operators to compare for equality (see Table 1-2).

TABLE 1-2 Equality operators

Name	Operator	Behavior
Equal to	==	True if the left-hand operand is equal to the right-hand operand
Not equal to	!=	True if the left-hand operand is not equal to the right-hand operand

These can be used between numeric variables and strings, but they should not be used when testing floating point (`float` and `double`) values as the nature of number storage on a computer means that values of these types are not held exactly. A program should not test to see if two floating point values are equal, instead it should subtract one from the other and determine if the absolute value of the result is less than a given tolerance value.

Logical values can be combined using logical operators (see Table 1-3).

TABLE 1-3 Logical operators

Name	Operator	Behavior
And	&	True if the left-hand operand and the right-hand operand are true
Or	|	True if the left-hand operand or the right-hand operand (or both) is/are true
Exclusive Or	^	True if the left-hand operand is not the same as the right-hand operandcc

The `&` and `|` operators have *conditional* versions: `&&` and `||`. These are only evaluated until it can be determined whether the result of the expression is true or false. In the case of `&&`, if the first operand is `false`, the program will not evaluate the second operand since it is already established that the result of the expression is `false`. In the case of `||`, if the first operand is `true` the second operand will not be evaluated as it is already established that the result of the expression is `true`. This is also referred to as *short circuiting* the evaluation of the expression. Listing 1-61 shows how this works. It contains instrumented methods that are called during the evaluation of the expression.

LISTING 1-61 Logical expressions

Click here to view code image

```
using System;

namespace LISTING_1_61_logical_expressions
{
    class Program
    {
        static int mOne()
        {
            Console.WriteLine("mOne called");
            return 1;
        }

        static int mTwo()
        {
            Console.WriteLine("mTwo called");
            return 2;
        }

        static void Main(string[] args)
        {
            if (mOne() == 2 && mTwo() == 1)
                Console.WriteLine("Hello world");

            Console.ReadKey();
        }
    }
}
```

When the program runs, it only outputs a message from the method `mOne`. The method `mTwo` is never called, even though it is in the expression. This is because the condition involving the value returned by `mOne` evaluates to `false`, which means there is no need to call `mTwo`. It is standard practice to use the conditional versions of the logical operators because it can improve program performance. If methods called in the logical expressions have side effects (which is bad programming practice) it may cause data-dependent faults to occur.

The switch construction

The `switch` construction let's a program use a value to select one of a followed by an expression that controls the switch. At run time the program will look for a matching value on a particular `case` clause, which identifies the code to be executed for that value. The code controlled by the `case` continues until a `break` statement, which marks the end of that clause. A switch can contain a `default` clause, identifying a clause to be performed if the control value doesn't match any case.

Listing 1-62 shows how the switch construction is used. The user specifies which of three commands they want to invoke by entering a number. If there is no matching command for the value that the user enters, the program prints out a message.

LISTING 1-62 The switch construction

Click here to view code image

```
using System;

namespace LISTING_1_62_The_switch_construction
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter command: ");
            int command = int.Parse(Console.ReadLin

            switch(command)
            {
                case 1:
                    Console.WriteLine("Command 1 ch
                    break;
                case 2:
                    Console.WriteLine("Command 2 ch
                    break;
                case 3:
                    Console.WriteLine("Command 3 ch
                    break;
                default:
                    Console.WriteLine("Please enter
                    break;
            }
            Console.ReadKey();
        }
    }
}
```

The `switch` construction will switch on character, string and enumerated values, and it is possible to group cases, as shown in Listing 1-63, which allows a user to select a command by entering a string. Note that the string that is entered is converted into lower case, and that both a long form (save) and a short form of the command (s) can be used.

LISTING 1-63 Switching on strings

Click here to view code image

```
Console.Write("Enter command: ");
string commandName = Console.ReadLine().ToLower() ;

switch (commandName)
{
    case "save":
    case "s":
        Console.WriteLine("Save command");
        break;
    case "load":
    case "l":
        Console.WriteLine("Load command");
        break;
    case "exit":
    case "e":
        Console.WriteLine("Exit command");
        break;
    default:
        Console.WriteLine("Please enter save, load
        break;
}
```

In C# it is not permissible for a program to "fall through" from the end of one case clause into another. Each clause must be explicitly ended with a `break`, a `return`, or by the program throwing an exception.

Switches are a nice example of a "luxury" C# feature in that they don't actually make something possible that you can't do any other way. You can create C# programs without using any switch statements, but they make it easier to create code that selects one behavior based on the value in a control variable.

Evaluating expressions

C# expressions are comprised of *operators* and *operands*. Operators specify the action to be performed, and are either literal values (for example the number 99) or variables. You have seen examples of operators and operands in use in the logical expressions that control the

Monadic operators are either *prefix* (given before the operand) or postfix (given after the operand). Alternatively, an operand can work on two (*binary*), or in the case of the conditional operator ?: three (ternary) operands.

The *context* of the use of an operator determines the actual behavior that the operator will exhibit. For example, the addition operator can be used to add two numeric operands together, or concatenate two strings together. The use of an incorrect context (for example adding a number to a string) will be detected by the compiler and cause a compilation error.

Each operator has a *priority* or *precedence* that determines when it is performed during expression evaluation. This precedence can be overridden by the use of parenthesis; elements enclosed in parenthesis are evaluated first. Operators also have an *associability*, which gives the order (left to right or right to left) in which they are evaluated if a number of them appear together. Listing 1-64 shows expression evaluation in action.

LISTING 1-64 Expression evaluation

Click here to view code image

```
int i = 0; // create i and set to 0

// Monadic operators - one operand
i++; // monadic ++ operator increment - i now 1
i--; // monadic -- operator decrement - i now 0

// Postfix monadic operator - perform after value g
Console.WriteLine(i++); // writes 0 and sets i to 1
// Prefix monadic operator - perform before value g
Console.WriteLine(++i); // writes 2 and sets i to 2

// Binary operators - two operands
i = 1 + 1; // sets i to 2
i = 1 + 2 * 3; // sets i to 7 because * performed f
i = (1 + 2) * 3; // sets i to 9 because + performed

string str = "";

str = str + "Hello"; // + performs string addition

// ternary operators - three operands
i = true ? 0 : 1; // sets i to 0 because condition 
```

Full details of precedence and associability for C# operators can be found on the web page: *https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/operators*.

SKILL 1.4: CREATE AND IMPLEMENT EVENTS AND CALLBACKS

You have seen that a program flows from statement to statement, processing data according to the statements that are performed. In early computers a program would flow from beginning to end, starting with the input data and producing some output before stopping. However, modern applications are not structured in this way. A large application will be made up of a large number of cooperating components which pass messages from one to another.

For example, one component of an application could be tasked with getting commands from the user. When the input component receives a valid user request, it passes this request into another component for processing. Each component can be created and tested individually before being integrated into the solution.

In order to create solutions that work in this way we need a mechanism by which one component can send a message to another. C# provides *events* to achieve this. In this section we discover the C# language features that provide event management, and then explore how these are used in modern application design.

This section covers how to:

- Create event handlers
- subscribe to and unsubscribe from events
- use built-in delegate types to create events
- create delegates
- use lambda expressions and anonymous methods

Event handlers

In the days before `async` and `await` were added to the C# language, a program would be forced to use events to manage asynchronous operations. Before initiating an asynchronous task, such as fetching a web page from a server, a program would need to bind a method to an event that would be generated when the action was complete. Today, events are more frequently used for inter-process communication.

We have seen that an object can provide a service for other objects by method exposed by the `Console` to display messages to the user of the program.

Events are used in the reverse of this situation, when you want an object to notify another object that something has happened. An object can be made to *publish* events to which other objects can *subscribe*. Components of a solution that communicate using events in this way are described as *loosely coupled*. The only thing one component has to know about the other is the design of the publish and subscribe mechanism.

Delegates and events

To understand how events are implemented you have to understand the concept of a C# *delegate*. This is a piece of data that contains a reference to a particular method in a class. When you take your car for a service you give the garage attendant your phone number so they can call you when your car is ready to be picked up. You can think of a delegate as the "phone number" of a method in a class. An event publisher is given a delegate that describes the method in the subscriber. The publisher can then call that delegate when the given event occurs and the method will run in the subscriber.

Action delegate

The .NET libraries provide a number of pre-defined delegate types. In the section, "Create Delegates," you will discover how to create your own delegate types.

There are a number of pre-defined `Action` delegate types. The simplest `Action` delegate represents a reference to a method that does not return a result (the method is of type `void`) and doesn't accept any parameters. You can use an `Action` to create a binding point for subscribers.

Listing 1-64 shows how an `Action` delegate can be used to create an event publisher. It contains an `Alarm` class that publishes to subscribers when an alarm is raised. The event `Action` delegate is called `OnAlarmRaised`. A process interested in alarms can bind subscribers to this event. The `RaiseAlarm` method is called in the alarm to raise the alarm. When `RaiseAlarm` runs it first checks to see if any subscriber methods have been bound to the `OnAlarmRaised` delegate. If they have, the delegate is called.

LISTING 1-65 Publish and subscribe

Click here to view code image

```
using System;

namespace LISTING_1_64_Publish_and_subscribe
{
    class Alarm
    {
        // Delegate for the alarm event
        public Action OnAlarmRaised { get; set; }

        // Called to raise an alarm
        public void RaiseAlarm()
        {
            // Only raise the alarm if someone has
            // subscribed.
            if (OnAlarmRaised != null)
            {
                OnAlarmRaised();
            }
        }
    }

    class Program
    {
        // Method that must run when the alarm is ra
        static void AlarmListener1()
        {
            Console.WriteLine("Alarm listener 1 cal
        }

        // Method that must run when the alarm is ra
        static void AlarmListener2()
        {
            Console.WriteLine("Alarm listener 2 cal
        }

        static void Main(string[] args)
        {
            // Create a new alarm
            Alarm alarm = new Alarm();

            // Connect the two listener methods
            alarm.OnAlarmRaised += AlarmListener1;
            alarm.OnAlarmRaised += AlarmListener2;

            // raise the alarm
            alarm.RaiseAlarm();
            Console.WriteLine("Alarm raised");

            Console.ReadKey();
        }
    }
}
```

Click here to view code image

```
Alarm listener 1 called
Alarm listener 2 called
Alarm raised
```

Event subscribers

Subscribers bind to a publisher by using the += operator. The += operator is *overloaded* to apply between a delegate and a behavior. It means "add this behavior to the ones for this delegate." The methods in a delegate are not guaranteed to be called in the order that they were added to the delegate. You can find out more about overloading in the "Create Types" section.

Delegates added to a published event are called on the same thread as the thread publishing the event. If a delegate blocks this thread, the entire publication mechanism is blocked. This means that a malicious or badly written subscriber has the ability to block the publication of events. This is addressed by the publisher starting an individual task to run each of the event subscribers. The `Delegate` object in a publisher exposes a method called `GetInvokcationList`, which can be used to get a list of all the subscribers. You can see this method in use later in the "Exceptions in event subscribers" section.

You can simplify the calling of the delegate by using the *null conditional* operator. This only performs an action if the given item is not null.

```
OnAlarmRaised?.Invoke();
```

The null conditional operator ".?" means, "only access this member of the class if the reference is not null." A delegate exposes an `Invoke` method to invoke the methods bound to the delegate. The behavior of the code is the same as that in Listing 1-64, but the code is shorter and clearer.

Unsubscribing from a delegate

You've seen that the += operator has been overloaded to allow methods to bind to events. The -= method is used to unsubscribe from events. The program in Listing 1-66 binds two methods to the alarm, raises the alarm, unbinds one of the methods, and raises the alarm again.

LISTING 1-66 Unsubscribing

Click here to view code image

```
static void Main(string[] args)
{
    // Create a new alarm
    Alarm alarm = new Alarm();

    // Connect the two listener methods
    alarm.OnAlarmRaised += AlarmListener1;
    alarm.OnAlarmRaised += AlarmListener2;

    alarm.RaiseAlarm();
    Console.WriteLine("Alarm raised");

    alarm.OnAlarmRaised -= AlarmListener1;
    alarm.RaiseAlarm();
    Console.WriteLine("Alarm raised");

    Console.ReadKey();
}
```

This program outputs the following:

Click here to view code image

```
Alarm listener 1 called
Alarm listener 2 called
Alarm raised
Alarm listener 2 called
```

If the same subscriber is added more than once to the same publisher, it will be called a corresponding number of times when the event occurs.

Using events

The `Alarm` object that we've created is not particularly secure. The `OnAlarmRaised` delegate has been made `public` so that subscribers can connect to it. However, this means that code external to the `Alarm` object can raise the alarm by directly calling the `OnAlarmRaised` delegate. External code can overwrite the value of `OnAlarmRaised`, potentially removing subscribers.

C# provides an `event` construction that allows a delegate to be specified as an *event*. This is shown in Listing 1-67. The keyword event is added before the definition of the delegate. The member `OnAlarmRaised` is now created as a data *field* in the `Alarm` class, rather than a *property*. `OnAlarmRaised` no longer has `get` or `set` behaviors. However, it is now not possible for code external to the `Alarm` class to assign values to `OnAlarmRaised`, and the `OnAlarmRaised` delegate can only be called from within the class where it is declared. In other words, adding the event keyword turns a delegate into a properly useful event.

```
class Alarm
{
    // Delegate for the alarm event
    public event Action OnAlarmRaised = delegate {}

    // Called to raise an alarm
    public void RaiseAlarm()
    {
            OnAlarmRaised();
    }
}
```

The code in Listing 1-67 above has one other improvement over previous versions. It creates a delegate instance and assigns it when `OnAlarmRaised` is created, so there is now no need to check whether or not the delegate has a value before calling it. This simplifies the `RaiseAlarm` method.

Create events with built-in delegate types

The event delegates created so far have used the `Action` class as the type of each event. This will work, but programs that use events should use the `EventHandler` class instead of `Action`. This is because the `EventHandler` class is the part of .NET designed to allow subscribers to be given data about an event. `EventHandler` is used throughout the .NET framework to manage events. An `EventHandler` can deliver data, or it can just signal that an event has taken place. Listing 1-68 shows how the `Alarm` class can use an `EventHandler` to indicate that an alarm has been raised.

LISTING 1-68 EventHandler alarm

Click here to view code image

```
class Alarm
{
    // Delegate for the alarm event
    public event EventHandler OnAlarmRaised = deleg

    // Called to raise an alarm
    // Does not provide any event arguments
    public void RaiseAlarm()
    {
        // Raises the alarm
        // The event handler receivers a reference 
        // raising this event
        OnAlarmRaised(this, EventArgs.Empty);
    }
}
```

The `EventHandler` delegate refers to a subscriber method that will accept two arguments. The first argument is a reference to the object raising the event. The second argument is a reference to an object of type `EventArgs` that provides information about the event. In the Listing 1-68 the second argument is set to `EventArgs.Empty`, to indicate that this event does not produce any data, it is simply a notification that an event has taken place.

The signature of the methods to be added to this delegate must reflect this. The `AlarmListener1` method accepts two parameters and can be used with this delegate.

Click here to view code image

```
private static void AlarmListener1(object sender, E
{
    // Only the sender is valid as this event doesn
    Console.WriteLine("Alarm listener 1 called");
}
```

Use EventArgs to deliver event information

The `Alarm` class created in Listing 1-68 allows a subscriber to receive a notification that an alarm has been raised, but it doesn't provide the subscriber with any description of the alarm. It is useful if subscribers can be given information about the alarm. Perhaps a string describing the location of the alarm would be useful.

You can do this by creating a class that can deliver this information and then use an `EventHandler` to deliver it. Listing 1-69 shows the `AlarmEventArgs` class, which is a sub class of the `EventArgs` class, and adds a `Location` property to it. If more event information is required, perhaps the date and time of the alarm, these can be added into the `AlarmEventArgs` class.

LISTING 1-69 EventHandler data

Click here to view code image

```
class AlarmEventArgs : EventArgs
{
    public string Location { get; set; }
```

```
            Location = location;
        }
    }
```

You now have your own type that can be used to describe an event which has occurred. The event is the alarm being raised, and the type you have created is called `AlarmEventAgs`. When the alarm is raised we want the handler for the alarm event to accept `AlarmEventArgs` objects so that the handler can be given details of the event.

The `EventHandler` delegate for the `OnAlarmRaised` event is declared to deliver arguments of type `AlarmEventArgs`. When the alarm is raised by the `RaiseAlarm` method the event is given a reference to the alarm and a newly created instance of `AlarmEventArgs` which describes the alarm event.

Click here to view code image

```
class Alarm
{
    // Delegate for the alarm event
    public event EventHandler<AlarmEventArgs> OnAla

    // Called to raise an alarm
    public void RaiseAlarm(string location)
    {
        OnAlarmRaised(this, new AlarmEventArgs(loca
    }
}
```

Subscribers to the event accept the `AlarmEventArgs` and can use the data in it. The method `AlarmListener1` below displays the location of the alarm that it obtains from its argument.

Click here to view code image

```
static void AlarmListener1(object source, AlarmEven
{
    Console.WriteLine("Alarm listener 1 called");
    Console.WriteLine("Alarm in {0}", args.Location
}
```

Note that a reference to the same `AlarmEventArgs` object is passed to each of the subscribers to the `OnAlarmRaised` event. This means that if one of the subscribers modifies the contents of the event description, subsequent subscribers will see the modified event. This can be useful if subscribers need to signal that a given event has been dealt with, but it can also be a source of unwanted side effects.

Exceptions in event subscribers

You now know how events work. A number of programs can *subscribe* to an event. They do this by binding a delegate to the event. The delegate serves as a reference to a piece of C# code which the subscriber wants to run when the event occurs. This piece of code is called an event handler.

In our example programs the event is an alarm being triggered. When the alarm is triggered the event will call all the event handlers that have subscribed to the alarm event. But what happens if one of the event handlers fails by throwing an exception? If code in one of the subscribers throws an uncaught exception the exception handling process ends at that point and no further subscribers will be notified. This would mean that some subscribers would not be informed of the event.

To resolve this issue each event handler can be called individually and then a single aggregate exception created which contains all the details of any exceptions that were thrown by event handlers. Listing 1-70 shows how this is done. The `GetInvocationList` method is used on the delegate to obtain a list of subscribers to the event. This list is then iterated and the `DynamicInvoke` method called for each subscriber. Any exceptions thrown by subscribers are caught and added to a list of exceptions. Note that the exception thrown by the subscriber is delivered by a `TypeInvocationException`, and it is the inner exception from this that must be saved.

LISTING 1-70 Aggregating exceptions

Click here to view code image

```
public void RaiseAlarm(string location)
{
    List<Exception> exceptionList = new List<Except

    foreach (Delegate handler in OnAlarmRaised.GetI
    {
        try
        {
            handler.DynamicInvoke(this, new AlarmEv
        }
        catch (TargetInvocationException e)
        {
            exceptionList.Add(e.InnerException);
        }
    }
```

The subscribers in Listing 1-70 both throw exceptions.

Click here to view code image

```
static void AlarmListener1(object source, AlarmEven
{
    Console.WriteLine("Alarm listener 1 called");
    Console.WriteLine("Alarm in {0}", args.Location
    throw new Exception("Bang");
}

static void AlarmListener2(object source, AlarmEven
{
    Console.WriteLine("Alarm listener 2 called");
    Console.WriteLine("Alarm in {0}", args.Location
    throw new Exception("Boom");
}
```

These can be caught and dealt with when the event is raised. The code below raises the alarm in the kitchen location, catches any exceptions that are thrown by subscribers to the alarm event, and then prints out the exception description.

Click here to view code image

```
try
{
    alarm.RaiseAlarm("Kitchen");
}
catch(AggregateException agg)
{
    foreach(Exception ex in agg.InnerExceptions)
        Console.WriteLine(ex.Message);
}
```

When this sample program runs it outputs the following. Note that the exceptions are listed after the subscriber methods have completed.

Click here to view code image

```
Alarm listener 1 called
Alarm in Kitchen
Alarm listener 2 called
Alarm in Kitchen
Bang
Boom
Alarm raised
```

Create delegates

Up until now we have used the `Action` and `EventHandler` types, which provide pre-defined delegates. We can, however, create our own delegates. Up until now the delegates that we have seen have maintained a collection of method references. Our applications have used the `+=` and `-=` operators to add method references to a given delegate. You can also create a delegate that refers to a single method in an object.

A `delegate` type is declared using the `delegate` keyword. The statement here creates a delegate type called `IntOperation` that can refer to a method of type integer that accepts two integer parameters.

```
delegate int IntOperation(int a, int b);
```

A program can now create delegate variables of type `IntOperation`. When a delegate variable is declared it can be set to refer a given method. In Listing 1-71 below the `op` variable is made to refer first to a method called `Add`, and then to a method called `Subtract`. Each time that `op` is called it will execute the method that it has been made to refer to.

LISTING 1-71 Create delegates

Click here to view code image

```
using System;

namespace LISTING_1_71_Create_delegates
{
    class Program
    {
        delegate int IntOperation(int a, int b);

        static int Add(int a, int b)
        {
            Console.WriteLine("Add called");
            return a + b;
        }

        static int Subtract(int a, int b)
        {
            Console.WriteLine("Subtract called");
            static void Main(string[] args)
            {
                // Explicitly create the delegate
                op = new IntOperation(Add);
                Console.WriteLine(op(2, 2));

                // Delegate is created automatically
                // from method
                op = Subtract;
                Console.WriteLine(op(2, 2));
                Console.ReadKey();
            }
        }
    }
```

Note that the code in Listing 1-71 also shows that a program can explicitly create an instance of the delegate class. The C# compiler will automatically generate the code to create a delegate instance when a method is assigned to the delegate variable.

Delegates can be used in exactly the same way as any other variable. You can have lists and dictionaries that contain delegates and you can also use them as parameters to methods.

Delegate vs delegate

It is important to understand the difference between `delegate` (with a lower-case d) and `Delegate` (with an upper-case D). The word `delegate` with a lower-case d is the keyword used in a C# program that tells the compiler to create a delegate type. It is used in Listing 1-71 to create the delegate type `IntOperation`.

```
delegate int IntOperation(int a, int b);
```

The word `Delegate` with an upper-case D is the abstract class that defines the behavior of delegate instances. Once the `delegate` keyword has been used to create a delegate type, objects of that delegate type will be realized as `Delegate` instances.

```
IntOperation op;
```

This statement creates an `IntOperation` value called `op`. The variable `op` is an instance of the `System.MultiCastDelegate` type, which is a child of the `Delegate` class. A program can use the variable `op` to either hold a collection of subscribers or to refer to a single method.

Use lambda expressions (anonymous methods)

Delegates allow a program to treat behaviors (methods in objects) as items of data. A delegate is an item of data that serves as a reference to a method in an object. This adds a tremendous amount of flexibility for programmers. However, delegates are hard work to use. The actual delegate type must first be declared and then made to refer to a particular method containing the code that describes the action to be performed.

Lambda expressions are a pure way of expressing the “something goes in, something happens and something comes out” part of behaviors. The types of the elements and the result to be returned are inferred from the context in which the lambda expression is used. Consider the following statement.

Click here to view code image

```
delegate int IntOperation(int a, int b);
```

This statement declares the `IntOperation` delegate that was used in Listing 1-71. The `IntOperation` delegate can refer to any operation that takes in two integer parameters and returns an integer result. Now consider this statement, which creates an `IntOperation` delegate called `add` and assigns it to a lambda expression that accepts two input parameters and returns their sum.

Click here to view code image

```
IntOperation add = (a, b) => a + b;
```

The operator => is called the *lambda operator*. The items a and `b` on the left of the lambda expression are mapped onto method parameters defined by the delegate. The statement on the right of the lambda expression gives the behavior of the expression, and in this case adds the two parameters together.

When describing the behavior of the lambda expression you can use the phrase “goes into” to describe what is happening. In this case you could say “a and b go into a plus b.” The name lambda comes from *lambda calculus*, a branch of mathematics that concerns “functional abstraction.”

This lambda expression accepts two integer parameters and returns an integer. Lambda expressions can accept multiple parameters and contain multiple statements, in which case the statements are enclosed in a block. Listing 1-72 shows how to create a lambda expression that prints out a message as well as performing a calculation.

LISTING 1-72 Lambda expressions

```
add = (a,b) =>
{
    Console.WriteLine("Add called");
    return a + b;
};
```

Closures

The code in a lambda expression can access variables in the code around it. These variables must be available when the lambda expression runs, so the compiler will extend the lifetime of variables used in lambda expressions.

Listing 1-73 shows how this works. The method `SetLocal` declares a local variable called `localInt` and sets its value to 99. Under normal circumstances the variable `localInt` would be destroyed upon completion of the `SetLocal` method. However, the `localInt` variable is used in a lambda expression, which is assigned to the delegate `getLocal`. The compiler makes sure that the `localInt` variable is available for use in the lambda expression when it is subsequently called from the `Main` method. This extension of variable life is called a *closure*.

LISTING 1-73 Closures

Click here to view code image

```
using System;

namespace LISTING_1_73_Closures
{
    class Program
    {
        delegate int GetValue();

        static GetValue getLocalInt;

        static void SetLocalInt()
        {
            // Local variable set to 99
            int localInt = 99;

            // Set delegate getLocalInt to a lambda
            // returns the value of localInt
            getLocalInt = () => localInt;
        }

        static void Main(string[] args)
        {
            SetLocalInt ();
            Console.WriteLine("Value of localInt {0
            Console.ReadKey();
        }
    }
}
```

Built in types for use with lambda expressions

Consider the following three statements:

Click here to view code image

```
delegate int IntOperation(int a, int b);
IntOperation add = (a, b) => a + b;
Console.WriteLIne(add(2,2));
```

The first statement creates a delegate called `IntOperation` that accepts two integer values and returns an integer result. The second statement creates an `IntOperation` called `add` which uses a lambda expression to describe what it does, which is to add the two parameters together and return the result. The third statement actually uses the `add` operation to calculate and print 2+2.

This works, but we had to create the `IntOperation` delegate type to specify a behavior that accepts two integers and returns their sum before we could create something that referred to a lambda expression of that type. There are a number of "built-in" delegate types that we can use to provide a context for a lambda expression.

The `Func` types provide a range of delegates for methods that accept values and return results. Listing 1-74 shows how the `Func` type is used to create an `add` behavior that has the same return type and parameters as the `IntOperation` delegate in Listing 1-71. There are versions of the `Func` type that accept up to 16 input items. The `add` method here accepts two integers and returns an integer as the result.

LISTING 1-74 Built in delegates

Click here to view code image

```
Func<int,int,int> add = (a, b) => a + b;
```

If the lambda expression doesn't return a result, you can use the `Action` type that you saw earlier when we created our first delegates. The statement below creates a delegate called `logMessage` that refers to a lambda expression that accepts a string and then prints it to the console

Click here to view code image

```
static Action<string> logMessage = (message) => Con
```

The `Predicate` built in delegate type lets you create code that takes a value of a particular type and returns true or false. The `dividesByThree` predicate below returns `true` if the value is divisible by 3.

Click here to view code image

```
Predicate<int> dividesByThree = (i) => i % 3 == 0;
```

Anonymous methods

Up until now we have been using lambda expressions that are attached to delegates. The delegate provides a name by which the code in the lambda expression can be accessed. However, a lambda expression can also be used directly in a context where you just want to express a particular behavior. The program in Listing 1-75 uses `Task.Run` to start a new task. The code performed by the task is expressed directly as a lambda expression, which is given as an argument to the `Task.Run` method. At no point does this code ever have a name.

LISTING 1-75 Lambda expression task

Click here to view code image

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace LISTING_1_75_lambda_expression_task
{
    class Program
    {
        static void Main(string[] args)
        {
            Task.Run( () =>
           {
               for (int i = 0; i < 5 ; i++)
               {
                   Console.WriteLine(i);
                   Thread.Sleep(500);
               }
           });

            Console.WriteLine("Task running..");
            Console.ReadKey();
        }
    }
}
```

A lambda expression used in this way can be described as an *anonymous method*; because it is a piece of functional code that doesn't have a name.

SKILL 1.5 IMPLEMENT EXCEPTION HANDLING

The need to handle errors is a natural consequence of writing useful programs. Exceptions are, as their name implies, errors that occur in exceptional circumstances. I don't consider something like an invalid user input as a situation that should be managed by the use of exceptions. User error is to be expected and dealt with in the normal running of a solution.

A program should use an exception to manage an error if it is not meaningful for the program to continue execution at the point the error occurs. For example if a network connection fails or a storage device becomes full. In this section you will learn how to deal with exceptions and how to create and manage exception events of your own.

This section covers how to:

- Handle exception types, including SQL exceptions, network exceptions, communication exceptions, network timeout exceptions
- Use catch statements
- Use base class of an exception
- Implement try-catch-finally blocks
- Throw exceptions
- Rethrow an exception
- Create custom exceptions
- Handle inner exceptions
- Handle aggregate exceptions

A program can indicate an error condition by *throwing* an exception object that may be *caught* by an *exception handler*. The purpose of the exception handler is to mitigate the effect of the exception. Handing an exception may involve such actions as alerting the user (if any), creating a log entry, releasing resources, and perhaps even shutting down the application in a well-managed way.

If an exception is not caught by an exception handler within the program it will be caught by the .NET environment and will cause the thread or task to terminate. Uncaught exceptions may cause threads and tasks to fail silently with no message to the user. An uncaught exception thrown in the foreground thread of an application will cause the application to terminate. Exceptions can be nested. When an exception is thrown, the .NET runtime searches up the call stack to find the "closest" exception handler to deal with the exception. This is a comparatively time-consuming process, which means that exceptions should not be used for "run of the mill" errors, but only invoked in exceptional circumstances.

A particular exception is described by an object, which is passed to the exception handler element of a program. The parent type of all exception objects is the `Exception` class. There are many exception types that describe particular error conditions. Start by considering exceptions that are thrown when a program uses elements of the .NET libraries.

- Input/output exceptions (`IOException`) are thrown during input/output operations.
- SQL (Structured Query Language) exceptions (`SqlException`) are thrown in response to an invalid SQL query. The SQL exception object contains a list of SqlError items that describe the error that occurred. In the case of an SQL exception generated by a LINQ query, the actual exception will not be thrown during the execution of the statement containing the query expression. It will be produced within the code that is iterating through the result returned by the expression. LINQ only begins evaluating the query when the results are requested.
- Communications exceptions (`CommunicationsException`) are thrown during Windows Communication Framework (WCF) operations and network timeout exceptions (`TimeOutExceptions`) are thrown when a network operation takes too long to compete.

As we shall see in the next section, it is possible to specifically handle a particular exception type by adding a matching catch clause. The signature of a given method does not indicate whether or not the method will generate any exceptions, so when using methods a programmer should carefully check the method documentation to determine whether or not exception handling is required.

The try-catch construction

Exception handling is performed by placing code to be protected in a block following the `try` keyword. This block of code can be followed by exception handler code, which is preceded by a `catch` keyword. The exception handler runs in the event of an exception being thrown. Listing 1-76 shows how this works. It contains a program that reads a string of text from the user and uses the `Parse` method from the `int` class to convert the text into an integer. This method will throw an exception if it can't convert the string into a valid integer.

LISTING 1-76 Try catch

Click here to view code image

```
using System;

namespace LISTING_1_76_try_catch
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.Write("Enter an integer: ")
                string numberText = Console.ReadLin
                int result;
                result = int.Parse(numberText);
                Console.WriteLine("You entered {0}"
            }
            catch
            {
                Console.WriteLine("Invalid number e
            }
            Console.ReadKey();
        }
    }
}
```

The program in Listing 1-76 does not use the exception object that is produced when the exception is thrown. The program in Listing 1-77 catches the exception and uses the `Message` and `StackTrace` properties of the exception to generate an error message.

LISTING 1-77 Exception object

Click here to view code image

```
        Console.Write("Enter an integer: ");
        string numberText = Console.ReadLine();
        int result;
        result = int.Parse(numberText);
        Console.WriteLine("You entered {0}", result);
    }
    catch(Exception ex)
    {
        Console.WriteLine("Message: " + ex.Message);
        Console.WriteLine("Stacktrace: " + ex.StackTrac
        Console.WriteLine("HelpLink: " + ex.HelpLink);
        Console.WriteLine("TargetSite: " + ex.TargetSit
        Console.WriteLine("Source:" + ex.Source);
    }
```

The `catch` keyword is followed by the type of the exception to be caught and the name to be used to refer to the exception object during the exception handler. In Listing 1-77 the value of `ex` is set to refer to the exception that is generated by the `Parse` method if it fails. If the user enters an invalid number, the output of the program will be listed as shown next. The first line of the output gives the error message and the `StackTrace` gives the position in the program at which the error occurred. The `HelpLink` property can be set to give further information about the exception. The `TargetSite` property gives the name of the method that causes the exception, and the `Source` property gives the name of the application that caused the error, or the name of the assembly if the application name has not been set. The output here shows the result of the exception.

Click here to view code image

```
Enter an integer: fred
Message: Input string was not in a correct format.
Stacktrace: at System.Number.StringToNumber(String
NumberBuffer& number, NumberFormatInfo info, Boolea
   at System.Number.ParseInt32(String s, NumberStyl
   at System.Int32.Parse(String s)
   at LISTING_1_77_Exception_object.Program.Main(St
HelpLink:
TargetSite: Void StringToNumber(System.String, Syst
Source:mscorlib
```

Using the base class of an exception

The example in Listing 1-77 catches all exceptions because the `Exception` class is the base class of all exception types. The program in Listing 1-78 reads an integer and then performs an integer division, dividing 1 by the entered number. This action can generate two exceptions; the user might not enter a valid number or the user might enter the value 0, which will cause a divide by zero exception. The program also contains a third catch element, which will catch any other exceptions that might be thrown by this code. The order of the catch elements is important. If the first catch element caught the `Exception` type the compiler would produce the error "A previous catch clause already catches all exceptions of this or of a super type ('Exception')". You must put the most abstract exception type last in the sequence.

LISTING 1-78 Exception types

Click here to view code image

```
using System;

namespace LISTING_1_78_Exception_types
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.Write("Enter an integer: ")
                string numberText = Console.ReadLin
                int result;
                result = int.Parse(numberText);
                Console.WriteLine("You entered {0}"
                int sum = 1 / result;
                Console.WriteLine("Sum is {0}", sum
            }
            catch (NotFiniteNumberException nx)
            {
                Console.WriteLine("Invalid number")
            }
            catch (DivideByZeroException zx)
            {
                Console.WriteLine("Divide by zero")
            }
            catch (Exception ex)
            {
                Console.WriteLine("Unexpected excep
            }

            Console.ReadKey();
        }
```

Note that not all arithmetic errors will throw an exception at this point in the code; if the same division is performed using the floating point or double precision type, the result will be evaluated as “infinity.”

Implement try-catch-finally blocks

The `try` construction can contain a `finally` element that identifies code that will be executed irrespective of whatever happens in the `try` construction. The program in Listing 1-79 displays the thank you message, whether or not any exceptions are thrown when the program runs.

LISTING 1-79 The finally block

Click here to view code image

```
using System;

namespace LISTING_1_79_The_finally_block
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.Write("Enter an integer: ")
                string numberText = Console.ReadLin
                int result;
                result = int.Parse(numberText);
                Console.WriteLine("You entered {0}"
                int sum = 1 / result;
                Console.WriteLine("Sum is {0}", sum
            }
            catch (NotFiniteNumberException nx)
            {
                Console.WriteLine("Invalid number")
            }
            catch (DivideByZeroException zx)
            {
                Console.WriteLine("Divide by zero")
            }
            catch (Exception ex)
            {
                Console.WriteLine("Unexpected excep
            }
            finally
            {
                Console.WriteLine("Thanks for using
            }

            Console.ReadKey();
        }
    }
}
```

Note that code in this block is guaranteed to run irrespective of what happens during the `try` construction. This includes situations where the code in the construction returns to a calling method or code in the exception handlers cause other exceptions to be thrown. The `finally` block is where a program can release any resources that it may be using.

The only situation in which a `finally` block will *not* be executed are:

- If preceding code (in either the try block or an exception handler) enters an infinite loop.
- If the programmer uses the `Environment.FailFast` method in the code protected by the `try` construction to explicitly request that any `finally` elements are ignored.

Throwing exceptions

A program can create and throw its own exceptions by using the `throw` statement to throw an exception instance. The `Exception` object constructor accepts a string that is used to deliver a descriptive message to the exception handler. The program in Listing 1-80 throws an exception and catches it, displaying the message.

LISTING 1-80 Throwing an exception

Click here to view code image

```
using System;

namespace LISTING_1_80_Throwing_an_exception
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                throw new Exception (
                          "I think you should know 
            }
            catch(Exception ex)
            }
            Console.ReadKey();
        }
    }
}
```

Rethrowing an exception

One of the fundamental principles of exception design is that catching an exception should not lead to errors being hidden from other parts of an application. Sometimes an exception will be caught that needs to be "passed up" to an enclosing exception handler. This might be because the low-level handler doesn't recognize the exception, or it might be because a handler at a higher level must also be alerted to the occurring exception. An exception can be re-thrown by using the keyword `throw` with no parameter:

```
throw;
```

You might think that when re-throwing an exception, you should give the exception object to be re-thrown, as shown here:

Click here to view code image

```
            catch(Exception ex)
            {
                Console.WriteLine(ex.Message);
                throw ex; // this will not preserve
            }
```

This is bad practice because it will remove the stack trace information that is part of the original exception and replace it with stack trace information that describes the position reached in the exception handler code. This will make it harder to work out what is going on when the error occurs, because the location of the error will be reported as being in your handler, rather than at the point at which the original exception was generated.

Inner exceptions

Another way to manage the propagation of error conditions in an application is to pass the original exception object as an *inner exception* for a higher-level exception handler to deal with. The `Exception` class contains an `InnerException` property that can be set when the exception is constructed. The constructor for the newly created exception is given a reference to the original exception.

Click here to view code image

```
            catch(Exception ex)
            {
                Console.WriteLine(ex.Message);
                throw new Exception("Something bad 
            }
```

Handler inner exceptions are covered in the "Handling Inner Exceptions" section.

Creating custom exceptions

When designing an application, you should also decide (and design) how the application will respond to any error conditions. This can include the creation of custom exception types for your program. The name of the exception class should end with "Exception." The `CalcException` class in Listing 1-81 contains an error code value that is set when the exception is constructed. This error code can then be used in the exception handler.

LISTING 1-81 Custom exceptions

Click here to view code image

```
using System;

namespace LISTING_1_81_Custom_exceptions
{
    class CalcException : Exception
    {
        public enum CalcErrorCodes
        {
            InvalidNumberText,
            DivideByZero
        }

        public CalcErrorCodes Error { get; set; }

        public CalcException(string message, CalcEr
        {
            Error = error;
        }
    }

    class Program
    {
            try
            {
                throw new CalcException("Calc faile
                CalcException.CalcErrorCodes.Invali
            }
            catch (CalcException ce)
            {
                Console.WriteLine("Error: {0}", ce.
            }
            Console.ReadKey();
        }
    }
}
```

Conditional clauses in catch blocks

An exception handler can re-throw an exception if it is not in a position to deal with the exception. You saw this in the "Throwing exceptions" section earlier in this chapter. An alternative to re-throwing an exception is to create a handler that only catches exceptions that contain particular data values.

The code in Listing 1-82 shows how this is achieved. The `when` keyword is followed by a conditional clause that performs a test on the exception object. The exception handler will only trigger in the event of an exception being thrown that has an `Error` property set to `DivideByZero`. An exception with any other error code is ignored, and in the case of the example program, will cause the program to terminate immediately as an unhandled exception has been thrown.

LISTING 1-82 Conditional clauses

Click here to view code image

```
try
{
    throw new CalcException("Calc failed", CalcExce
}
catch (CalcException ce) when (ce.Error == CalcExce
{
    Console.WriteLine("Divide by zero error");
}
```

If the program in Listing 1-82 is executed it will display the message: "Divide by zero error." If the error value in the `throw` statement is changed from `DivideByZero` to `InvalidNumberText`, the program will instead fail with an unhandled exception.

This mechanism is more efficient than re-throwing an exception, because the .NET runtime doesn't have to rebuild the exception object prior to re-throwing it.

Handling inner exceptions

An exception can contain an inner exception property that is set when it is constructed. An exception handler can extract this and use it as part of the exception management process. The program in Listing 1-83 contains an exception handler that throws a new exception containing an inner exception that describes the error. If the user enters text that cannot be parsed into an integer, an exception is thrown that is caught, and then a new exception is raised with the error, "Calculator Failure." The new exception contains the original exception as an inner exception. The new exception is then caught and displayed.

LISTING 1-83 Inner exceptions

Click here to view code image

```
using System;

namespace LISTING_1_83_Inner_exceptions
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                try
                {
                    Console.Write("Enter an integer
                    string numberText = Console.Rea
                    int result;
                    result = int.Parse(numberText);
                }
                catch (Exception ex)
                {
                    throw new Exception("Calculator
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                Console.WriteLine(ex.InnerException
                Console.WriteLine(ex.InnerException
            }
        }
    }
```

Note that the use of inner exceptions must be planned, because handlers must be expecting the inner exception to refer to meaningful information about the error that has occurred.

Handling aggregate exceptions

Some .NET exception types contain lists of inner exceptions. These are called “aggregate exceptions.” They occur when more than one thing can fail as an operation is performed, or when the results of a series of actions need to be brought together.

The program in Listing1-84 shows a situation in which aggregate exceptions are used to deliver results from a method that is called to read the text from a web page. The `AgregateException` is caught and the message from each exception is displayed.

LISTING 1-84 Aggregate exceptions

Click here to view code image

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

namespace LISTING_1_84_Aggregate_exceptions
{
    class Program
    {
        async static Task<string> FetchWebPage(stri
        {
            var httpClient = new HttpClient();
            var response = await httpClient.GetAsyn
            return await response.Content.ReadAsStr
        }

        static void Main(string[] args)
        {
            try
            {
                Task<string> getpage = FetchWebPage
                getpage.Wait();
                Console.WriteLine(getpage.Result);
            }
            catch ( AggregateException ag)
            {
                foreach(Exception e in ag.InnerExce
                {
                    Console.WriteLine(e.Message);
                }
            }
            Console.ReadKey();
        }
    }
}
```

You have also seen the use of aggregate exceptions when considering how to deal with exceptions thrown by parallel LINQ enquiries (LISTING_1_10_Exceptions_in_PLINQ) and when aggregating exceptions raised during event handers (LISTING_1_70_Aggregating_exceptions).

Exceptions as part of managed error handing

You should make sure that your code deals with any exceptions that may be thrown by methods, propagating the exception onward if necessary to ensure that exception events are not hidden from different parts of your application.

Your methods should also throw exceptions in situations where it is not meaningful to continue with an action. If a method throws an exception the caller must deal with that exception if the program is to continue running. However, if a method returns an error code when something goes wrong, this error code could be ignored by the caller.

It is important that you consider how to manage error conditions during the design of an application. It is very hard to add error management to an application once it has been created.

THOUGHT EXPERIMENTS

In these thought experiments, demonstrate your skills and knowledge of the topics covered in this chapter. You can find the answers to these thought experiments in the next section.

1 Using multiple tasks

At the design stage of an application you will have to decide whether/how to use tasks and threads. Here are some questions that you might like to consider:

1. Given the difficulties in synchronization and management, is it worth the effort to implement applications using multiple tasks?
2. Is it still worth the effort to use multiple-tasks if you only have one

3. What kind of applications benefit the most from the use of multi-tasking applications?

4. Are there situations when you really should not use multi-tasking?

5. I need a background process that is going to compress a large number of data files. Should I use a task or a thread?

6. What is the difference between the `WaitAll` and `WaitAny` method when waiting for a large number of tasks to complete?

7. What is the difference between a continuation task and a child task?

8. What is the difference between the `WaitAll` and `WhenAll` methods?

9. What happens when a method call is awaited?

10. What is special about a concurrent collection?

2 Managing multithreading

Multithreaded applications will need to give special consideration to the way that data is processed and the ordering of operations. They will also have to contain provision for process management.

Here are some questions that you might like to consider:

1. Will program errors caused by a poor multithreading implementation always manifest themselves as faults in an application?

2. Does the fact that the processor is suddenly at 100% loading indicate that two processes are stuck in a deadly embrace?

3. Could you make an object thread safe by enclosing the body code of all the methods in lock statements to make all the method actions atomic?

4. If you're not sure about potential race conditions, is it best to add lock statements around critical sections of your code "just in case?"

5. Should a task always generate an exception if it is cancelled?

6. Could you make an application that automatically cancelled deadlocked processes?

3 Program flow

The C# language provides constructions that the programmer can use to manage the flow of program execution. Statements can be repeated a given number of times, until a specific state has been reached, or over a collection of items. Statements can be executed conditionally using the `if` construction, or selected on the basis of a particular control variable in the case of a switch.

Here are some questions which you might like to consider:

1. Is it necessary to have both the while and the do-while looping construction?

2. Is a break statement the only way of exiting a loop before it completes?

3. Can you identify a situation where you would use a `for` loop to iterate through a collection rather than a `foreach` loop?

4. The and (`&`) logical operator and the or (`|`) logical operator have "short circuit" versions (`&&` and `||`), which only evaluate elements until it can be determined whether or not a given expression is true or false. Why does the exclusive-or (`^`) operator not have a short circuit version?

5. Is it true that each C# operator has a behavior that is the same for every context in which it is used?

6. Can you always be certain of the precise sequence of operations when an expression is evaluated?

4 Events and callbacks

Programmers can use delegates to create applications from "loosely coupled" objects that communicate by message events. An object can subscribe to events published by another object which exposes a delegate.

A lambda expression can be used to create "anonymous functions" that exists as pure behaviors. Delegates can be made to refer to anonymous functions.

Here are some questions that you might like to consider:

1. How does a delegate actually work?

2. What happens if a delegate is assigned to a delegate?

3. What happens if a delegate is assigned to itself?

4. Is there an upper limit for the number of subscribers that a publisher delegate can have?

5. What does the lambda operator (`=>`) actually do?

6. Can code in a lambda expression access data from the enclosing code?

5 Exceptions

design custom exception types and create an error handling strategy that manages how exceptions are propagated and managed. It is important to put exceptions in the correct perspective; any error that you would expect the system to deal with in the normal course of events should not be handled by the use of an exception, they should be reserved for situations where it is not meaningful for the application to continue.

Here are some questions that you might like to consider:

1. Is a method receiving a date of "31st of February" something that should cause an exception?
2. Should I make sure that all exceptions are always caught by my program?
3. Can you return to the code after an exception has been caught?
4. Why do we need the finally clause? Can't code to be run after the code in a try clause just be put straight after the end of the catch?
5. Should my application always use custom exceptions?

THOUGH EXPERIMENT ANSWERS

This section provides the solutions for the tasks included in the thought experiments.

1 Using multiple tasks

1. Breaking an application into separate tasks is not just something that you would do to make use of the performance advantages. Tasks provide a very useful level of abstraction. Once the communication between the tasks has been designed, each one can be worked on independently. Partitioning an application into tasks can be seen as a natural extension of component-based design.
2. Consider a program that is performing a numeric analysis of a particular data set. Once the data has been loaded into memory the time it takes to be processed is determined by the speed of the CPU. We would say that this program was *cpu bound*. If a system only has one processor, converting the program to use a multi-threaded implementation would not improve the rate at which the data is processed. However, consider a program that is continuously interacting with the file system. We would say this application was *IO (input/output) bound*, in that the rate at which it can work is determined by performance of the file system. When an IO bound program is active it will frequently be unable to do any work as it waits file transactions to complete. A multi-threaded solution would be able to perform processing at the times when one task was waiting for an input/output operation to complete.
3. A CPU bound application will benefit from a multi-tasking approach if the host system has more than one CPU. Note that the programmer may have to spend some effort re-working the application to allow it to work over multiple tasks. An application that contains a mix of CPU bound and IO bound tasks can benefit most from a multi-threading approach as there will always be something for the processor to do.
4. The asynchronous nature of a multi-threaded application makes it very hard to guarantee that a given operation will be completed within a specified time. Threads can be given priority levels, but these are not hard and fast. Applications with critical timing requirements should not be implemented using multiple threads.
5. When choosing between tasks and threads you need to consider what you want to achieve. Tasks are good for background processes. Threads can be used for background processes but can also run as foreground processes. Tasks are easier to create. In this situation a Task would make the most sense, because it would be easier to create and will never need to operate in the foreground.
6. The `WaitAll` method will return when all the tasks that are running have completed. The `WaitAny` method will return when the first of the running tasks completes.
7. A continuation task is a task that is started when an existing task completes. A child task is created by a parent task and executes independently of the parent.
8. The `WaitAll` method accepts a list of Task references as an argument and returns when all the tasks have completed. The `WaitAll` method does not return a result. The `WhenAll` method accepts a list of `Task` references as an argument and returns a `Task` that delivers the results returned by those tasks. The `WhenAll` method is can be awaited, which allows it to be used in asynchronous code.
9. Methods that can be awaited return a reference to a Task. When a program performs an `await` in an asynchronous method, the asynchronous method returns immediately. When the task returned by the awaited method completes the remainder of the asynchronous method will complete. A good example of a situation where this can be used to good effect is in user interfaces. A user interface method, for example the code that implements a behavior when a button is clicked in the user interface, must return as quickly as possible. Otherwise the user interface will become unresponsive. By using await the button click handler can complete and return, while the action initiated by the button click runs in parallel.
10. A concurrent collection is one which has behaviors that can be used

behaviors that allow processes to only perform updates to data stored in the collection when it is valid to do so.

2 Managing multithreading

1. There is no guarantee that the conditions that might cause a race condition to cause a problem will arise. Natural delays in a system might mean that events occur in an order that means race conditions do not cause a problem. For example, if one task is writing a file to a disk, there is a good chance that this task will take longer to complete than another which is updating the display. However, problems might suddenly appear when the user buys a faster disk-drive or installs more memory. This might speed up the write action and cause the application to become unstable. In fact, these symptoms are the best way of diagnosing problems like this. The first question you should ask a user whose program has suddenly become unreliable is "Have you made any changes to the underlying system?"

2. The simplest form of deadlock occurs when two tasks are waiting for locks to be released by each other. More complex deadlocks can be spread over a larger number of tasks. However, when a deadlock occurs a program is not executing very rapidly, in fact nothing is happening at all, all the tasks are waiting for each other, and the application is taking no processor time at all.

3. Making all method calls in an object atomic is a very heavy handed and dangerous approach to achieving thread safety. It may well make an application vulnerable to deadlocks and could also remove any performance advantage that would be gained by using multiple processors. Furthermore, it does not protect against issues caused by the use of reference parameters to a method.

4. Adding lock statements around critical pieces of code without thinking about what the code actually does is very dangerous. It may well lead to performance bottlenecks and deadlocks. It is very difficult to add thread safety to a program once it has been written. The only sure-fire way I've found of writing thread safe code is to design it that way from the start.

5. A task can be cancelled "silently", in which case it will just end. Alternatively, it can be made to generate an exception when cancelled. From a design perspective I tend to regard cancelling a task as an exceptional action and would therefore expect there to be some way of propagating this event through the system so that it can be registered in some way.

6. If you use the Monitor construction to manage entry into atomic actions it is possible for a task to retain control when it fails to get a lock and enter an atomic action. You can even establish a time-out by adding a timeout value to a call of the `TryEnter` method to try and enter an atomic action. It would therefore be possible for a program to recognize that a given lock had been unavailable for a more than a certain length of time. It could then use a cancellation token to try and stop a task that might have obtained the synchronization object. Of course, this would only work correctly if the task that is causing the deadlock is checking its cancellation token while it is waiting for its lock. In other words, yes it would be possible to do this, but it would entail a lot of extra work. The best way to avoid deadlocks is to make sure that they are not present in the design of your system.

3 Program flow

1. The while and do-while loop constructions differ in only one respect. The while construction will not execute any code if the control condition is false, whereas the do-while construction tests the control condition at the completion of the loop, so the code controlled by it is always executed once, even if the control condition is false at the start. At the expense of a slightly more complicated program, it is possible to only use one form of the loop construction, both are provided to make the programmers life slightly easier.

2. If a `break` is used in a loop the program will exit the loop at that point. It is also possible to exit a loop by returning from the method in which the loop is declared. Finally a program will exit a loop if it throws an exception within the loop.

3. A `foreach` construction provides a very convenient way to express a need to enumerate each of the items in a collection. However, each enumerated item is provided on a "read-only" basis. If the program wants to change the value of an enumerated item it is necessary to use a `for` loop to access each item. Note that if the `foreach` construction is iterating through a collection of reference types it is perfectly permissible to change properties of each iterated item.

4. There is no short-circuit version of the exclusive-or operator because it is not possible to determine whether an exclusive-or is true by just examining one of the operands. Remember that an exclusive-or operation returns true if the two operands are different. It is not possible to make this determination by just examining one of the operands.

5. No. C# operators have different behaviors depending on the *context* in which they are used. The integer and floating-point types are held in quite different ways in the memory of the computer. When arithmetic operators are used between these types the compiler will select the correct operator to be used depending on the type of the operands in the expression.

which the operations are performed is not something that a programmer can make any assumptions about.

4 Events and callbacks

1. A delegate is a type that can refer to a method in an object with a particular signature. A delegate can also contain lists of method references. A delegate is called in exactly the same way as the method it can refer to. If the delegate contains a list of method references, each method is called in no particular order. The base type of all delegate objects is the Delegate class.

2. Delegates can be assigned to other delegates, as long as they have the same signature. Delegates can also be added to the subscribers of a multi-cast delegate.

3. A delegate can be assigned to itself, but this is not a sensible thing to do. If the delegate is ever called the program will enter an infinite loop.

4. There is no upper limit, over and above the capacity of the data structure in the `Delegate` class which holds references to subscribers.

5. The lambda operator (=>) separates the items that go into the expression (the input parameters) from the statement or statement block that implements the behavior of the lambda expression.

6. Code in a lambda expression can access variables declared in code enclosing the lambda expression. Note that code in a lambda expression is not executed when the lambda expression is declared, a reference to the lambda expression may be followed at a much later time. The C# compiler will generate code that maintains variables used by the code in the lambda expression even if they would normally be out of scope at the time the code in the lambda expression is performed.

5 Exceptions

1. It all depends on the context of the action. If the date is being entered by the user, it is reasonable for the program to display an error message and request that the date be re-entered. In this case no exception will be thrown. However, if the date is being supplied to a method that is expected to create a transaction with this date, it is reasonable for that method to throw an exception if given an invalid date value, as it would not be meaningful for the method to generate anything. Throwing an exception, rather than returning an error code or a null value, is a way of ensuring that the program will fail in a properly noticeable way and maximizes the chances of the mistake being detected. From a design perspective, the need for exceptions in this situation would be completely removed if the method creating the transaction was passed a DateTime structure. This can only contain a valid date.

2. No. There is only one thing worse than a program that fails, and that is a program that fails silently. You might think it is a good idea to wrap your entire program in a try construction. This would mean that the program would never fail due to an uncaught exception. However, it would also make the program impossible to debug and prone to failing silently. You should take care to catch all the exceptions that you know how to deal with, and let the remaining ones pass through so that they are either caught by layers above (in a managed and designed way) or propagate as errors which can be caught and dealt with.

3. There is no way to return to the statements after an exception has been thrown. If a program wants to re-try code that may throw exceptions, this code must be placed in a loop.

4. We need a finally clause because code in the catch clause may return from a method or throw other exceptions, in which case the program will not reach the statements following the catch. The finally clause is guaranteed to be executed in all circumstances.

5. The standard Exception type provides a great deal of flexibility; it contains a Dictionary which can be filled with name-value pairs that describe an error. Custom exception types can be useful if none of the existing exception types are suitable for the exception context. Because different catch clauses can be allocated to different exception types, they also make exceptions easier to manage.

CHAPTER SUMMARY

- Multi-threading allows an application to be made up of a number of cooperating processes.

- When you create a multi-threaded application, its behaviors are spread amongst a number of co-operating tasks. The tasks may be performed in parallel if the host computer has multiple central processor units (CPUs). A single CPU will execute each active task in turn on a round-robin basis.

- Creating a multi-threaded application can be as simple as taking an existing set of actions and using the `Task.Parallel` library or Parallel LINQ to execute the actions as multiple tasks.

- The .NET Task class provides a high-level abstraction of a running task. Tasks can return values and continuation tasks can run automatically when one or more tasks complete. Tasks can also start child tasks, with the *parent* task being held until all the child tasks have completed.

background processes. Tasks can only run as background processes. A foreground process will run until complete, whereas a background process will be terminated upon the completion of the foreground process that created it. It is also possible to set the priority of a thread (although this is purely advisory, it doesn't guarantee that a given thread will be allowed a particular level of access to the processor).

- `Async` and `await` make it very easy to create multi-threaded applications. The `await` keyword precedes a call to a method that has been identified as asynchronous by the `async` keyword. An asynchronous method can return a Task (if the asynchronous method returns void) or a Task<type> (if the method wishes to return a value). The compiler will generate code that allows an asynchronous method to return to the caller when the `await` keyword is reached. The statements following the `await` will be performed concurrently. This way of working is of particular value when building the user interface to a program. An asynchronous action bound to a button press will unblock the user interface at the first `await`, rather than stopping the user interface until the action performed by the button has completed. An application can catch exceptions thrown during asynchronous calls by enclosing the awaited action in an exception handler, but this only works if the awaited action returns a value. For this reason, void asynchronous calls should be avoided as there is no way of determining whether they succeeded or not.

- The standard .NET collection classes are not *thread safe*. This means that if multiple tasks attempt to share data using a List, the contents of the list will be corrupted. The .NET framework provides a set of concurrent collections that can be shared by multiple active tasks. The `BlockingCollection` class provides a wrapper around `ConcurrentStack`, `ConcurrentQueue` and `ConcurrentBag` concurrent collections. Tasks using a `BlockingCollection` will be blocked (paused) if there is no room for additional items, or if they try to take items from an empty collection. The concurrent collections provide "try" versions of methods to extract items which return whether an action succeeded or not. This is because in the time between determining that there are items available and reading them it is possible that another task could have removed the items. The `ConcurrentDictionary` class provides additional methods for the conditional update of items in the dictionary.

- Multi-threaded applications are vulnerable to *race conditions* where actions by a task on a shared data item are not run to completion before that task is replaced by another.

- Race conditions can be addressed by making actions *atomic,* in that they will always complete before another task is allowed to perform the action. This is achieved by the use of lock objects. A task claims the lock and while it has the lock, other tasks trying to claim that lock and enter the atomic code will be blocked. This may result in a queue of blocked tasks waiting for a particular lock to be released.

- Two tasks waiting for locks from each other are said to be *deadlocked* or in a *deadly embrace*. Deadlocks can arise as a result of code which waits for access to a lock object while inside an atomic action. This should be addressed by good design.

- The Monitor mechanism for locking can be used in preference to the lock keyword if an application would benefit from tasks being able to determine whether or not they can have access to an atomic action, rather than being blocked as soon as they try to acquire the lock.

- Simple actions, such as updating a particular variable, can be achieved by using the interlocked operations, rather than creating atomic blocks of code.

- Variables that may be used by multiple processes can be marked as "volatile." This tells the compiler not to perform optimization such as caching the value of the variable in a processor register, or changing the order of instructions.

- Tasks can be cancelled by the use of cancellation tokens. As a task runs it must check state of the token to determine if a cancellation has been requested. This is an important difference between a Task and a Thread. Threads can be aborted by another process at any time. One task can request that another task be cancelled, but this will only actually result in that thread ending if the code in the task is checking the cancellation token. Note that this does not mean that an active task can never be removed from memory however, because tasks run in the background they are automatically terminated upon completion of the foreground process that created them.

- Methods in an object must be made *thread safe* if they are to be used in applications that contain multiple tasks. Access to data members of the class containing the thread safe method must be managed in an atomic manner. Parameters passed into the method by reference are vulnerable to changes to elements in the parameter that may occur while the method is running.

- A `while` construction is useful in situations where you want to repeat something as long as a condition is true. A do-while construction is useful in situations where you want to do something and then repeat it if the action failed.

- A for construction is an easy way to perform initialization, test and update actions on a loop. A for construction frequently involves the management of a counter value, but this is not the only way in

- A `foreach` construction can be used to enumerate the items in a collection. The collection provides a method that will provide an enumerator which is then iterated by the `foreach` construction. The items in the collection are provided as read-only.

- The `break` statement allows a program to exit a loop immediately. A large number of breaks in a loop can make it hard to discern the circumstances in which the loop exited. A loop can also be exited upon return from the method in which the loop is running and when code in the loop throws an exception that is not caught in the loop.

- The continue statement allows a program to return to the "top" of a loop and repeat the loop without going any further through the loop code. Note that any update and test behaviors will be performed when a continue statement runs.

- An if construction allows the conditional execution of a statement or block of statements. If constructions are controlled by a logical expression and can be followed by an else clause that identifies the statement to be performed if the condition is false. If conditions can be nested, an else portion of an `if` construction always binds to the "closest" `if`.

- Logical expressions evaluate to true or false. Variables can be compared using relational and equality operators. Logical values can be combined using `and`, or `and` exclusive or operators. The `and` operator and the or operator have "short circuit" versions that are only evaluated to the point where the value of the expression result can be determined.

- The `switch` construction allows the selective of a given behavior from the value of a control value which can be an integer, string or character. A default behavior can be specified if the control value does not match any of the selections.

- An expression contains operands and operators. The operands are literal values or variables. Operators have priority and associability that determine the point that the operator is applied during the evaluation of the expression.

- C# programs can use delegates to create variables that can serve as references to methods in objects. An object wishing to receive notifications from a publisher can use a delegate to specify a method to be called by the event publisher. A single publisher delegate makes calls to its subscribers, each of which has provided a delegate.

- The `event` keyword allows a delegate to be used in a secure way and the `EventArgs` classes describe a pattern that is used throughout .NET to allow events to deliver data into a subscriber.

- Delegates can also be used as references to individual methods. A delegate referring to a method can be regarded as a piece of data which describes an action.

- A lambda expression allows an action to be expressed directly and provides a convenient shorthand when writing code. The type of parameters and value returned by a lambda expression are inferred from the context of the call.

- An exception is thrown by a program to indicate a situation in which the program cannot continue normal operation. Execution is transferred from the statements being executed in a `try` block of code to an exception handler in a `catch` block, which was written to deal with the exception. An exception that is thrown in code that is not within a `try` construction will cause the executing thread to terminate.

- A `finally` element can be used in a try construction to specify code that will be always be executed.

- A program throws an exception by creating a new exception instance and then using the `throw` keyword to throw it. All exception objects are derived from the Exception class. There are a large number of exception types that are used in the .NET libraries to describe error conditions; a programmer can also create their own exception types that contain their own error-specific information.

- An exception object contains information that describes the error, including a "stack trace" that indicates the point in the program source where the exception is thrown. An exception can also contain an inner exception reference so that a new exception can be wrapped round one that has been caught, before passing the exception to another layer of exception management.

PREV
About the Author

NEXT
Chapter 2 Create and use t