Table of Contents (Detailed)

Chapter 1

.NET Fundamentals

.NET Fundamentals

Objectives

After completing this unit you will be able to:

- **Understand the problems Microsoft .NET is designed to solve.**
- **Understand the basic programming model of Microsoft .NET.**
- **Understand the basic programming tools provided by Microsoft .NET.**
- **Discuss .NET Native, .NET Core and cross-platform development.**

What Is Microsoft .NET?

- **Microsoft .NET was developed to solve three fundamental problems.**
- **First, the Microsoft Windows programming model must be unified to remove the widely varied programming models and approaches that exist among the various Microsoft development technologies.**
- **Second, Microsoft based solutions must be capable of interacting with the modern world of heterogeneous computing environments.**
- **Third, Microsoft needs a development paradigm that is capable of being expanded to encompass future development strategies, technologies, and customer demands.**

Open Standards and Interoperability

- **The modern computing environment contains a vast variety of hardware and software systems.**

  Computers range from mainframes and high-end servers, to workstations and PCs, and to small mobile devices such as PDAs and cell phones.

  Operating systems include traditional mainframe systems, many flavors of Unix including Android, Linux, Apple's iOS, several versions of Windows, real-time systems and more.

  Many different languages, databases, application development tools and middleware products are used.

- **Applications need to be able to work in this heterogeneous environment.**

  Even shrink-wrapped applications deployed on a single PC may use the Internet for registration and updates.

- **The key to interoperability among applications is the use of *standards*, such as HTML, HTTP, XML, SOAP, and TCP/IP.**

Windows Development Problems

- **In classic Windows development design and language choice often clashed.**
  - Visual Basic vs. C++ approach
  - IDispatch, Dual, or Vtable interfaces
  - VB vs. MFC
  - ODBC or OLEDB or ADO
- **Application deployment was hard.**
  - Critical entries in Registry for COM components
  - No versioning strategy
  - DLL Hell
- **Security was difficult to implement.**
  - No way to control code or give code rights to certain actions and deny it the right to do other actions.
  - Security model is difficult to understand. Did you ever pass anything but NULL to a LPSECURITY_ATTRIBUTES argument?
- **Too much time is spent in writing plumbing code that the system should provide.**
  - MTS/COM+ was a step in the right direction.

Common Language Runtime

- **The first step in solving the three fundamental problems is for Microsoft .NET to provide a set of underlying services available to all languages.**
- **The runtime environment provided by .NET that provides these services is called the *Common Language Runtime* or CLR.**

  A runtime provides services to executing programs.

  Traditionally there are different runtimes for different programming environments. Examples of runtimes include the standard C library, MFC, the Visual Basic 6 runtime and the Java Virtual Machine.

- **These services are available to all languages that follow the rules of the CLR.**

  C# and Visual Basic are examples of Microsoft languages that are fully compliant with the CLR requirements.

  Not all languages use all the features of the CLR.

- **As a terminology note, beginning with .NET 2.0, Microsoft has dropped the ".NET" in the Visual Basic language.**

  The pre-.NET version of the language is now referred to as Visual Basic 6 or VB6.

Serialization Example

- **Let us use serialization to illustrate how the CLR provides a set of services that unifies the Microsoft development paradigm.**

  Every programmer has to do it.

  It can get complicated with nested objects, complicated data structures, and a variety of data storages.

  The programmer should also be able to override the system service if necessary.

- **See the *Serialize* example in this chapter.**

Serialization Example (Cont'd)

- **Ignore the language details covered in a later chapter.**

```
[Serializable]
class Customer
{
   public string name;
   public long id;
}
class Test
{
   static void Main(string[] args)
   {
      ArrayList list = new ArrayList();

      Customer cust = new Customer();
      cust.name = "Charles Darwin";
      cust.id = 10;
      list.Add(cust);

      cust = new Customer();
      cust.name = "Isaac Newton";
      cust.id = 20;
      list.Add(cust);

      foreach (Customer x in list)
         Console.WriteLine(x.name + ": " + x.id);

      Console.WriteLine("Saving Customer List");
      FileStream s = new FileStream("cust.txt",
                                    FileMode.Create);
      SoapFormatter f = new SoapFormatter();
      SaveFile(s, f, list);
```

Serialization Example (Cont'd)

```
      Console.WriteLine("Restoring to New List");
      s = new FileStream("cust.txt",
         FileMode.Open);
      f = new SoapFormatter();
      ArrayList list2 =
         (ArrayList)RestoreFile(s, f);

      foreach (Customer y in list2)
         Console.WriteLine(y.name + ": " + y.id);
   }

   public static void SaveFile(Stream s,
   IFormatter f, IList list)
   {
      f.Serialize(s, list);
      s.Close();
   }

   public static IList RestoreFile(Stream s,
   IFormatter f)
   {
      IList list = (IList)f.Deserialize(s);
      s.Close();
      return list;
   }
}
```

Attribute-Based Programming

- **We add two Customer objects to the collection, and print them out. We save the collection to disk and then restore it. The identical list is printed out.**

```
Charles Darwin: 10
Isaac Newton: 20
Saving Customer List
Restoring to New List
Charles Darwin: 10
Isaac Newton: 20
Press enter to continue...
```

- **We wrote no code to save or restore the list!**

  We just annotated the class we wanted to save with the **Serializable** attribute.

  We specified the format (SOAP) that the data was to be saved.

  We specified the medium (disk) where the data was saved.

  This is typical class partitioning in the .NET Framework.

- **Attribute-based programming is used throughout .NET to describe how code and data should be treated by the framework.**

Metadata

- **The compiler adds the *Serializable* attribute to the *metadata* of the Customer class.**
- **Metadata provides the Common Language Runtime with information it needs to provide services to the application.**

  Version and locale information

  All the types

  Details about each type, including name, visibility, etc.

  Details about the members of each type, such as methods, the signatures of methods, etc.

  Attributes

- **Metadata is stored with the application so that .NET applications are self-describing. The registry is not used.**

  The CLR can query the metadata at runtime. It can see if the **Serializable** attribute is present. It can find out the structure of the Customer object in order to save and restore it.

Types

- ***Types* are at the heart of the programming model for the CLR.**

  Most of the **metadata** is organized by type.

- **A type is analogous to a class in most object-oriented programming languages, providing an abstraction of data and behavior, grouped together.**

- **A type in the CLR contains:**

  Fields (data members)

  Methods

  Properties

  Events (which are now full fledged members of the Microsoft programming paradigm).

NET Framework Class Library

- **The *SoapFormatter* and *FileStream* classes are two of the thousands of classes in the .NET Framework that provide system services.**
- **The functionality provided includes:**

  Base Class Library (basic functionality such as strings, arrays and formatting).

  Networking

  Security

  Remoting

  Diagnostics

  I/O

  Database

  XML

  Web Services

  Web programming

  Windows User Interface

- **This framework is usable by all CLR compliant languages.**

Interface-Based Programming

- **Interfaces allow you to work with abstract types in a way that allows for extensible programming.**
- **The *SaveFile* and *RestoreFile* routines are written using the *IList* and *IFormatter* interfaces.**
- **These routines will work with all the collection classes that support the *IList* interface, and the formatters that support the *IFormatter* interface.**
- **Implementation inheritance permits code reuse.**
- **You can implement the *ISerializable* interface to override the framework's implementation.**

  The metadata for the type tells the framework that the class has implemented the interface.

- **Interface-based programming allows classes to provide implementations of standard functionality that can be used by the framework.**

Everything is an Object

- **Every type in .NET derives from *System.Object*.**[1]
- **Every type, system or user defined, has metadata.**

  In the sample the framework can walk through the ArrayList of Customer objects and save each one as well as the array itself.

- **All access to objects in .NET is through object references.**

[1] An exception is the pointer type, which is rarely used in C#.

Common Type System

- **The Common Type System (CTS) defines the rules for the types and operations that the CLR will support.**

  The CTS limits .NET classes to single implementation inheritance.

  The CTS is designed for a wide range of languages, not all languages will support all features of the CTS.

- **The CTS makes it *possible* to guarantee type safety.**

  Access to objects can be restricted to object references (no pointers), each reference refers to a defined memory. Access to that layout is only through public methods and fields.

  By performing a local analysis of the class, you can verify to make sure that the code does not perform any inappropriate memory access. You do not have to analyze the users of the class.

- **.NET compilers emit Microsoft Intermediate Language (MSIL or IL) not native code.**

  MSIL is platform independent.

  Type-safe code can be restricted to a subset of verifiable MSIL expressions.

  Once code is verified, it is verified for all platforms.

ILDASM

- **The Microsoft Intermediate Language Disassembler (ILDASM) can display the metadata and MSIL instructions associated with .NET code.**

  It is a very useful tool both for debugging and increasing your understanding of the .NET infrastructure.

- **You may wish to add ILDASM to your Tools menu in Visual Studio 2015.**

  Use the command Tools | External Tools. Click the Add button, enter ILDASM for the Title, and click the ... button to navigate to the folder \Program Files\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6 Tools.

ILDASM (Cont'd)

- **You can use ILDASM to examine the .NET framework code.**

  Here is a fragment of the MSIL from the **Serialize** example.

.NET Framework SDK Tools

- **Installing Visual Studio 2015 will also install the .NET Framework SDK, version 10.0A.**

  These tools are located in the folder \Program Files\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6 Tools.

They can be run at the command line from the Visual Studio 2015 Command Prompt[2], which can be started from All Programs | Visual Studio 2015 | Visual Studio Tools | Developer's Command Prompt for VS2015.

[2] You may need to run the Command Prompt as Administrator in some cases.

Language Interoperability

- **Having all language compilers use a common intermediate language and common base class makes it *possible* for languages to interoperate.**

  All languages need not implement all parts of the CTS.

  One language can have a feature that another does not.

- **The *Common Language Specification* (CLS) defines a subset of the CTS that represents the basic functionality that all .NET languages should implement if they are to interoperate with each other.**

  For example, a class written in Visual Basic can inherit from a class written in C#.

  Interlanguage debugging is possible.

  CLS rule: Method calls need not support a variable number of arguments even though such a construct can be expressed in MSIL.

  CLS prohibits the use of pointers.

- **CLS compliance only applies to public features.**

  C# code should not define public and protected class names that differ only by case sensitivity since languages as Visual Basic are not case sensitive. Private C# fields could have such names.

Managed Code

- **In the serialization example we never freed any allocated memory.**

  Memory that is no longer referenced can be reclaimed by the CLR's garbage collector.

  Automatic memory management eliminates the common programming error of memory leaks.

  Garbage collection is one of the services provided to .NET applications by the Common Language Runtime.

- **Managed code uses the services of the CLR.**

  MSIL can express access to unmanaged data in legacy code.

- **Type-safe code cannot be subverted.**

  For example, a buffer overwrite is not able to corrupt other data structures or programs. Security policy can be applied to type-safe code.

- **Type-safe code can be secured.**

  Access to files or user interface features can be controlled.

  You can prevent the execution of code from unknown sources.

  You can prevent access to unmanaged code to prevent subversion of .NET security.

  Paths of execution of .NET code to be isolated from one another.

Assemblies

x .NET programs are deployed as an *assembly*.

The metadata about the entire assembly is stored in the assembly's manifest.

An assembly has one or more EXEs or DLLs with associated metadata information.

Assembly Deployment

- **The assemblies can be uniquely named.**

  Assemblies can be versioned and the version is part of the assembly's name.

  Unique (strong) names use a public/private encryption scheme.

  The culture used can also be made part of the assembly name.

- **Assemblies are self-describing. Information is in the metadata associated with the assembly, not in the System Registry.**

- **Private, or xcopy deployment requires only that all the assemblies an application needs are in the same directory.**

  This makes deployment of components much simpler.

- **Public assemblies require a strong name and an entry in the Global Assembly Cache (GAC).**

- **Either approach means the end of DLL Hell!**

  Components with different versions can be deployed side by side and need not interfere with each other.

JIT Compilation

- **Before executing on the target machine, MSIL is translated by a just-in-time (JIT) compiler to native code.**
- **Some code typically will never be executed during a program run.**

  Hence it may be more efficient to translate MSIL as needed during execution, storing the native code for reuse.

- **When a type is loaded, the loader attaches a stub to each method of the type.**

  On the first call the stub passes control to the JIT, which translates to native code and modifies the stub to save the address of the translated native code.

  On subsequent calls to the method transfer is then made directly to the native code.

- **As part of JIT compilation code goes through a verification process.**

  Type safety is verified, using both the MSIL and metadata.

  Security restrictions are checked.

ASP.NET and Web Services

- **.NET includes a totally redone version of the popular Active Server Pages technology, known as *ASP.NET*.**
- **Whereas ASP relied on interpreted script code interspersed with page formatting commands, ASP.NET relies on *compiled* code.**

  The code can be written in any .NET language, including C#, Visual Basic, JScript.NET and C++/CLI.

- **ASP.NET provides *Web Forms* which vastly simplifies creating Web user interfaces.**

  Drag and drop in Visual Studio 2015 makes it very easy to lay out forms.

  Also supported are ASP.NET MVC and Web API.

- **For application integration across the internet, *Web services* use the SOAP protocol.**

  The beautiful thing about a Web service is that from the perspective of a programmer, a Web service is no different from any other kind of service implemented by a class in a .NET language.

  Or you can use third-party web services which you did not know existed when you designed your application.

- **Web services and C# (or Visual Basic) as a scripting language allows Web programming to follow an object-oriented programming model.**

The Role of XML

- **XML is ubiquitous in .NET and is highly important in Microsoft's overall vision.**
- **Some uses of XML in .NET include:**

  XML is used for encoding requests and responses in the SOAP protocol.

  XML is the serialization format for disconnected datasets in ADO.NET.

  XML is used extensively in configuration files.

  XML documentation can be automatically generated by .NET languages.

  .NET classes provide a very convenient API for XML programming as an alternative to DOM or SAX.

Performance

- **Concerns about performance of managed code are similar to the concerns assembly language programmers had with high level languages.**
- **Garbage collection usually produces faster allocation than C++ unmanaged heap allocation. Deallocation is done on a separate thread by the garbage collector.**
- **JIT compilation takes a hit the first time when verification and translation take place, but subsequent executions pay no penalty.**
- **There is a penalty when security checks have to be made that require a stack walk.**
- **Compiled ASP.NET code is going to be a lot faster than interpreted ASP pages.**
- **Bottom line: for most of the code that is written, any small loss in performance is far outweighed by the gains in reliability and ease of development.**

  High performance servers might still have to use technologies such as ATL Server and C++.

- **Apps targeting the Windows 10 platform may use .NET Native to achieve higher performance.**

.NET Native

- **.NET Native is a precompilation technology for building and deploying apps to Windows 10.**

  Rather than generating IL, Windows apps are compiled directly to native code for faster startup and execution.

- **.NET Native changes the way in which .NET Framework applications are built and executed.**

  During precompilation required portions of the .NET Framework are statically linked to your code, enabling the compiler to perform global code optimization.

  The .NET Native runtime is optimized for static precompilation.

  .NET Native uses the same backend as the C++ compiler for superior performance.

- **.NET Native brings the performance of C++ to managed code.**

- **See MSDN for more information:**

https://msdn.microsoft.com/en-us/library/dn584397(v=vs.110).aspx

.NET Core

- **.NET Core is a modular subset version of the .NET Framework that is portable across multiple platforms.**
- **Rather than one large assembly, .NET Core is released through NuGet in smaller feature-specific assembly packages.**
- **.NET Core provides key functionality used in applications regardless of platform.**

  This common functionality provides for shared code that can be used across platforms.

  Your application then links in additional platform-specific code.

- **Microsoft platforms you can target include traditional desktop Windows and Windows phones.**
- **Through third-party tools such as Xamian you can target Android and iOS.**
- **Visual Studio 2015 provides support for cross-platform development.**

Summary

- .NET solves problems of past Windows development.
- One development paradigm for all languages exists.
- Design and programming language no longer conflict.
- Web services provide an API for applications across the Internet, typically using the SOAP protocol.
- SOAP supports a high degree of interoperability, since it is based on widely adopted standards such as HTTP and XML.
- .NET has many features which will create a much more robust Windows operating system.
- .NET uses managed code with services provided by the Common Language Runtime that uses the Common Type System.
- The .NET Framework is a very large class library available consistently across many languages.
- Deployment is more rational and includes a versioning strategy.
- Metadata, attribute-based security, code verification, and type-safe assembly isolation make developing secure applications much easier.
- Plumbing code for fundamental system services is there, yet you can extend it or replace it if necessary.

Chapter 7

.NET Threading

.NET Threading

Objectives

After completing this unit you will be able to:

- **Use the *Thread* class to implement multithreading in .NET applications.**
- **Use the *Monitor* class to program safe concurrent access to shared data.**
- **Use the *ThreadPool* class to obtain threads from a pool that is managed by the system.**
- **Describe the difference between foreground and background threads.**
- **Describe different classes that can be used for synchronizing threads.**
- **Use the Task Parallel Library to implement task parallelism and data parallelism in .NET applications.**

Threads

- **Operating systems use processes to separate the different applications that they are executing. Threads run inside of processes to allow for multiple execution paths inside of a process.**
- **Threads are what are scheduled by the operating system, not processes or application domains.**

  Threads maintain a context, exception handlers catch exceptions thrown within the thread in which they occur.

  Machine registers and stack are also part of the thread's context.

  This context has to be saved when the operating system's scheduler switches from one thread to another.

  The **Thread** object that represents the current executing thread can be found from the static property **Thread.CurrentThread**.

.NET Threading Model

- **The .NET Framework provides extensive support for multiple thread programming in the *System.Threading* namespace.**
- **The core class is *Thread*, which encapsulates a thread of execution.**

  This class provides methods to start and suspend threads, to sleep, and to perform other thread management functions.

- **The method that will execute for a thread is encapsulated inside a delegate of type *ThreadStart*.**

  Recall that a delegate can wrap either a static or an instance method.

- **When starting a thread, it is frequently useful to define an associated class, which will contain instance data for the thread, including initialization information.**

  A designated method of this class can be used as the **ThreadStart** delegate method.

- **.NET 4 introduced the Task Parallel Library (TPL) to simplify the implementation of parallel code using multiple threads.**

  We will discuss TPL later in the chapter after covering the fundamentals of threads in .NET.

Console Log Example

- **The *ThreadDemo* program provides an illustration of this architecture.**

  The **ConsoleLog** class encapsulates a thread ID and parameters specifying a sleep interval and a count of how many lines of output will be written to the console.

  It also provides a **Stopwatch** object (System.Diagnostics namespace) to provide timings.

  It provides the method **ConsoleLog** that writes out logging information to the console, showing the thread ID and number of elapsed (millisecond) ticks. Here is the program code:

```
using System;
using System.Diagnostics;
using System.Threading;

class ConsoleLog
{
   private int delta;
   private int count;
   private int ticks = 0;
   public static Stopwatch stopWatch =
      new Stopwatch();
   public ConsoleLog(int delta, int count)
   {
      this.delta = delta;
      this.count = count;
   }
...
```

Console Log Example (Cont'd)

```
   ...
   public void ConsoleThread()
   {
      for (int i = 0; i < count; i++)
      {
         Console.WriteLine(
            "Thread {0}: ticks = {1}",
            Thread.CurrentThread.ManagedThreadId,
            ticks);
         Thread.Sleep(delta);
         ticks += delta;
      }
      Console.WriteLine(
         "Thread {0} is terminating",
         Thread.CurrentThread.ManagedThreadId);
      Console.WriteLine(
         "\nElapsed time:\n\t" +
         ConsoleLog.stopWatch.Elapsed);
   }
}

public class ThreadDemo
{
   public static void Main()
   {
        Sequential();
        UseThreads();
   }
   ...
}
```

Console Log Example (Cont'd)

```
...
   public static void Sequential()
   {
      Console.WriteLine("Sequential");
      ConsoleLog.stopWatch.Restart();
      ConsoleLog slowLog = new ConsoleLog(1000, 5);
      ConsoleLog fastLog = new ConsoleLog(400, 5);
      slowLog.ConsoleThread();
      fastLog.ConsoleThread();
   }
   public static void UseThreads()
   {
      Console.WriteLine("\nUsing Threads");
      ConsoleLog.stopWatch.Restart();
      ConsoleLog slowLog = new ConsoleLog(1000, 5);
      ConsoleLog fastLog = new ConsoleLog(400, 5);
      ThreadStart slowStart =
         new ThreadStart(slowLog.ConsoleThread);
      ThreadStart fastStart =
         new ThreadStart(fastLog.ConsoleThread);
      Thread slowThread = new Thread(slowStart);
      Thread fastThread = new Thread(fastStart);
      Console.WriteLine("Starting threads ...");
      slowThread.Start();
      fastThread.Start();
      Console.WriteLine("Threads have started");
   }
}
```

Console Log Example (Cont'd)

- **The program is configured with a "slow" thread and a "fast" thread.**

  The slow thread will sleep for 1 second between outputs, and the fast thread will sleep for only 400 milliseconds. A **ConsoleLog** object is created for each of these threads, initialized with appropriate parameters. Both will do five lines of output.

- **Next, appropriate delegates are created of type *ThreadStart*.**

  Notice that we use an instance method, **ConsoleThread**, as the delegate method.

  Use of an instance method rather than a static method is appropriate in this case, because we want to associate parameter values (sleep interval and output count) with each delegate instance.

- **We then create and start the threads.**

  We write a message to the console just before and just after starting the threads.

  You will notice a slight delay as the program executes, reflecting the sleep periods.

- **Notice the difference in total time from the stopwatch for the method using threading versus the sequential version.**

Race Conditions

- **A major issue in concurrency is shared data.**

  If two computations access the same data, different results can be obtained depending on the timing of the different accesses, a situation known as a race condition.

  Race conditions present a programming challenge because they can occur unpredictably. Careful programming is required to ensure they do not occur.

- **Race conditions can easily arise in multithreaded applications, because threads belonging to the same process share the same address space and thus can share data.**

- **Consider two threads making deposits to a bank account, where the deposit operation is not atomic:**

  Get balance.

  Add amount to balance.

  Store balance.

Race Condition Example

- **The following sequence of actions will then produce a race condition, with invalid results.**

  1. Balance starts at $100.
  2. Thread 1 makes deposit of $25 and is interrupted after getting balance and adding amount to balance, but before storing balance.
  3. Thread 2 makes deposit of $5000 and goes to completion, storing $5100.
  4. Thread 1 now finishes, storing $125, overwriting the result of thread 2. The $5000 deposit has been lost!

- **The program *ThreadAccount\Race* illustrates this race condition.**

  The **Account** class has a method **DelayDeposit**, which updates the balance non-atomically.

  The thread sleeps for 5 seconds in the middle of the update operation, leaving open a window of vulnerability for another thread to come in.

Race Condition Example (Cont'd)

```
using System.Threading;

public class Account
{
   protected decimal balance;
   public Account (decimal balance)
   {
      this.balance = balance;
   }
   public void Deposit(decimal amount)
   {
      balance += amount;
   }
   public void DelayDeposit(decimal amount)
   {
      decimal newbal = balance + amount;
      Thread.Sleep(5000);
      balance = newbal;
   }
   public decimal Balance
   {
      get
      {
         return balance;
      }
   }
}
```

Race Condition Example (Cont'd)

- **The test program launches threads in a manner similar to that used in the *ThreadDemo* program.**

  The **AsynchAccount** class contains the thread methods that will be used by thread 1 (to call **DelayDeposit**) and thread 2 (to call **Deposit**).

```
using System;
using System.Threading;

class AsynchAccount
{
   private decimal amount;
   public AsynchAccount(decimal amount)
   {
      this.amount = amount;
   }
   public void AsynchDelayDeposit()
   {
      ThreadAccount.account.DelayDeposit(amount);
   }
   public void AsynchDeposit()
   {
      ThreadAccount.account.Deposit(amount);
   }
}
...
```

Race Condition Example (Cont'd)

```
public class ThreadAccount
{
   public static Account account;
   public static void Main()
   {
      account = new Account(100);
      AsynchAccount asynch1 = new
         AsynchAccount(25);
      AsynchAccount asynch2 = new
         AsynchAccount(5000);
      ThreadStart start1 = new
         ThreadStart(asynch1.AsynchDelayDeposit);
      ThreadStart start2 = new
         ThreadStart(asynch2.AsynchDeposit);
      Console.WriteLine("balance = {0:C}",
         account.Balance);
      Console.WriteLine(
         "delay deposit of {0:C} on thread 1", 25);
      Thread t1 = new Thread(start1);
      Thread t2 = new Thread(start2);
      t1.Start();
      Console.WriteLine(
          "deposit of {0:C} on thread 2", 5000);
      t2.Start();
      t2.Join();
      Console.WriteLine(
"balance = {0:C} (thread 2 done)",account.Balance);
      t1.Join();
      Console.WriteLine(
"balance = {0:C} (thread 1 done)",account.Balance);
   }
}
```

- ***t2.Join*** **blocks current thread until thread *t2* finishes.**

  This technique enables us to show the balance after a thread has definitely completed.

Thread Synchronization

- **Such race conditions can be avoided by serializing access to the shared data.**
- **Suppose only one thread at a time is allowed to access the bank account.**

  Then the first thread that starts to access the balance will complete the operation before another thread begins to access the balance (the second thread will be blocked).

  In this case threads synchronize based on accessing data.

- **Another way threads can synchronize is for one thread to block until another thread has completed.**

  The **Join** method is a means for accomplishing this kind of thread synchronization, as illustrated above.

- **The *System.Threading* namespace provides a number of thread synchronization facilities.**

  We will illustrate use of the **Monitor** class.

Monitor

- **You can serialize access to shared data using the *Enter* and *Exit* methods of the *Monitor* class.**

  Monitor.Enter obtains the monitor lock for an object. An object is passed as a parameter. This call will block if another thread has entered the monitor of the same object. It will not block if the current thread has previously entered the monitor.

  Monitor.Exit releases the monitor lock. If one or more threads are waiting to acquire the lock, and the current thread has executed **Exit** as many times as it has executed **Enter**, one of the threads will be unblocked and allowed to proceed.

- **An object reference is passed as the parameter to *Monitor.Enter* and *Monitor.Exit*.**

  This is the object on which the monitor lock is acquired or released. To acquire a lock on the current object, pass **this**.

- **The program *ThreadAccount\Monitor* illustrates the use of monitors to protect the critical section where the balance is updated.**

- **The program *ThreadAccount\Lock* illustrates an alternative implementation using C# keyword *lock*.**

Monitor Example

```
using System;
using System.Threading;
public class Account {
   protected decimal balance;
   public Account (decimal balance)
   {
      this.balance = balance;
   }
   public void Deposit(decimal amount)
   {
      Monitor.Enter(this);
      balance += amount;
      Monitor.Exit(this);
      ShowBalance();
   }
   public void DelayDeposit(decimal amount)
   {
      Thread.Sleep(5000);
      Monitor.Enter(this);
      balance += amount;
      Monitor.Exit(this);
      ShowBalance();
   }
   public decimal Balance
   {
      get
      {
         return balance;
      }
   }
   private void ShowBalance()
   {
   Console.WriteLine("balance = {0:C} ({1})",
       balance, Thread.CurrentThread.Name);
   }
}
```

Using C# *lock* Keyword

```
public class Account
{
   protected decimal balance;
   protected string owner;
   public Account (decimal balance)
   {
      this.balance = balance;
      this.owner = "Tom Thread";
   }
   public void Deposit(decimal amount)
   {
      lock(this)
      {
         balance += amount;
      }
         ShowBalance();
   }
   public void DelayDeposit(decimal amount)
   {
      Thread.Sleep(5000);
      lock(this)
      {
         balance += amount;
      }
      ShowBalance();
   }
   ...
}
```

Synchronization of Collections

- **Some lists, such as *TraceListeners* are thread safe. When this collection is modified, a copy is modified and the reference is set to the copy.**
- **Normally, collections like *ArrayList* are not thread safe. Making them automatically thread safe would decrease the performance of the collection even when thread safety is not an issue.**
- **An *ArrayList* has a static *Synchronized* method to return a thread-safe version of the *ArrayList*. The *IsSynchronized* property indicates if the *ArrayList* is thread safe or not.**

  The **SyncRoot** property can return an object that can be used to synchronize access to a collection.

- **The *System.Collections.Concurrent* namespace, introduced with .NET 4, provides several thread-safe collection classes.**

ThreadPool Class

- **The *ThreadPool* class provides a pool of worker threads that are managed by the system.**

  You are thus relieved of having to create and start your own thread.

- **The static method *QueueUserWorkItem()* will retrieve a thread from the thread pool, if available, and start it. If no thread is available, the request will be queued until a thread is available.**

```
public static bool QueueUserWorkItem(
    WaitCallback callBack
)
```

- **The *WaitCallback* delegate represents a callback method that that is to be executed on a ThreadPool thread.**

```
public delegate void WaitCallback(
    Object state
)
```

  You create the delegate by passing your callback method to the **WaitCallback** constructor.

- **ThreadPool threads are always background threads.**

ThreadPool Example

- **We illustrate the use of *ThreadPool* with another implementation of the Console Log example.**

  See **ThreadPoolDemo\Step1** in the chapter folder. It has the same structure as the earlier example where we created threads ourselves.

  To match the **WaitCallback** delegate, the thread procedure takes an **Object** input parameter.

```
class ThreadWithState
{
   private int Delta;
   private int Count;
   private int ticks = 0;

   public ThreadWithState(int delta, int count)
   {
      this.Delta = delta;
      this.Count = count;
   }
   public void ConsoleLog()
   {
     ...
   }
   public void ThreadProc(Object info)
   {
      ConsoleLog();
   }
}
```

Starting a ThreadPool Thread

- **Now we don't separately create a Thread object and start it.**

  We use a single call to **QueueUserWorkItem()**.

```
public static void Main()
{
   ThreadWithState slowLog =
      new ThreadWithState(1000, 5);
   ThreadWithState fastLog =
      new ThreadWithState(400, 5);

   // Queue the slow log on a background thread
   ThreadPool.QueueUserWorkItem(
      new WaitCallback(slowLog.ThreadProc));

   // Run the fast log on the main thread
   fastLog.ConsoleLog();
}
```

- **Build and run.**

```
Thread 1: ticks = 0
Thread 3: ticks = 0
Thread 1: ticks = 400
Thread 1: ticks = 800
Thread 3: ticks = 1000
Thread 1: ticks = 1200
Thread 1: ticks = 1600
Thread 1 is terminating
Thread 3: ticks = 2000
```

The fast thread finishes, and the program exits before the slow thread can finish. Why?

Foreground and Background Threads

- **Managed threads are either foreground threads or background threads.**
- **A background thread is identical to a foreground thread except that it does not keep the managed execution environment running.**

  When all foreground threads have stopped, the system stops all the background threads and shuts down.

- **Threads created from the *Thread* class are by default foreground threads.**

  You can make a thread a background thread by setting the **IsBackground** property to true.

- **Threads in the ThreadPool are always background threads, and this cannot be changed.**

Synchronizing Threads

- **To make your application using background threads behave properly, you need to synchronize the background threads with the foreground threads.**
- **The .NET Framework provides several useful classes that can be used for such synchronization:**

  EventWaitHandle

  AutoResetEvent

  ManualResetEvent

  CountdownEvent

- **For synchronizing a single thread with another, *AutoResetEvent* and *ManualResetEvent* are useful.**

  A thread blocks when calling **WaitOne()** until another thread calls **Set()**, which signals the wait handle.

  The difference between these two classes is that an **AutoResetEvent** automatically resets after it has been signaled and has released a single waiting thread.

- **A *CountdownEvent* is useful for synchronizing multiple threads with another thread.**

  It maintains a counter of the number of times it has been signaled and will release waiting threads when the counter has been decremented to zero.

Improved ThreadPool Example

- **Step2 of the *ThreadPoolDemo* program illustrates use of an *AutoResetEvent* to synchronize the worker thread with the main thread.**

```
class ThreadWithState
{
   ...
   public static AutoResetEvent
      ev = new AutoResetEvent(false);

   ...
   public void ThreadProc(Object info)
   {
      ConsoleLog();
      ev.Set();
   }
}
...
public static void Main()
{
   ThreadWithState slowLog =
      new ThreadWithState(1000, 5);
   ThreadWithState fastLog =
      new ThreadWithState(400, 5);

   // Queue the slow log on a background thread
   ThreadPool.QueueUserWorkItem(
      new WaitCallback(slowLog.ThreadProc));

   // Run the fast log on the main thread
   fastLog.ConsoleLog();

   ThreadWithState.ev.WaitOne();
}
```

Improved Example (Cont'd)

- **Build and run the Step2 version.**

```
Thread 1: ticks = 0
Thread 3: ticks = 0
Thread 1: ticks = 400
Thread 1: ticks = 800
Thread 3: ticks = 1000
Thread 1: ticks = 1200
Thread 1: ticks = 1600
Thread 3: ticks = 2000
Thread 1 is terminating
Thread 3: ticks = 3000
Thread 3: ticks = 4000
Thread 3 is terminating
```

- **Now the worker thread runs to completion before the application exits.**

Task Parallel Library (TPL)

- **The Task Parallel Library (TPL) provides classes and methods that simplify programming with multiple threads.**

  The **Task** class provides a wrapper for threads from the **ThreadPool**.

- **TPL supports two kinds of parallelism:**

  Task parallelism facilitates parallel program using *tasks*, which are like threads but at a higher level of abstraction.

  Data parallelism facilitates performing the same operation concurrently on elements in an array or collection.

- **Besides making parallel programming simpler, TPL can make programs more efficient.**

  TPL can scale the degree of concurrency dynamically based on the number of processors available.

Task Example

- **We illustrate the uses of tasks by providing another implementation of our console log example.**

  See **TaskDemo** in the chapter directory.

  The same wrapper class is used for encapsulating the thread procedure.

  The main program creates an array of **Task** objects.

```
static void Main(string[] args)
{
   ThreadWithState slowLog =
      new ThreadWithState(1000, 5);
   ThreadWithState mediumLog =
      new ThreadWithState(700, 5);
   ThreadWithState fastLog =
      new ThreadWithState(400, 5);

   // Array of tasks. Use a factory for
   // starting third task
   Task[] tasks = new Task[3];
   tasks[0] = new Task(() => slowLog.ThreadProc());
   tasks[0].Start();
   tasks[1] = Task.Run(
      () => mediumLog.ThreadProc());
   tasks[2] = Task.Factory.StartNew(
      () => fastLog.ThreadProc());

   // Wait for all tasks to complete
   Task.WaitAll(tasks);
}
```

Starting Tasks

- **There are three ways to start a task.**

  Instantiate a **Task** object and call the **Start()**method.

```
tasks[0] = new Task(() => slowLog.ThreadProc());
tasks[0].Start();
```

  Call the static **Run()** method to create and start the task in one operation

```
tasks[1] = Task.Run(
   () => mediumLog.ThreadProc());
```

  Call the static **StartNew()** method of the **Factory** property to create and start the task in one operation. This technique provides a greater variety of options than the **Run()** method.

```
tasks[2] = Task.Factory.StartNew(
   () => fastLog.ThreadProc());
```

- **In each case we pass in a delegate, which can be conveniently expressed by a lambda expression.**

Waiting for Task Completion

- **The *Task* class makes it easy to wait on multiple threads.**

  Use the **WaitAll()** method. You do not need to manually create synchronization objects. The system handles the synchronization for you.

```
// Wait for all tasks to complete
Task.WaitAll(tasks);
```

- **Here is the result of running the program:**

```
Thread 3: ticks = 0
Thread 5: ticks = 0
Thread 4: ticks = 0
Thread 5: ticks = 400
Thread 4: ticks = 700
Thread 5: ticks = 800
Thread 3: ticks = 1000
Thread 5: ticks = 1200
Thread 4: ticks = 1400
Thread 5: ticks = 1600
Thread 3: ticks = 2000
Thread 5 is terminating
Thread 4: ticks = 2100
Thread 4: ticks = 2800
Thread 3: ticks = 3000
Thread 4 is terminating
Thread 3: ticks = 4000
Thread 3 is terminating
```

Data Parallelism

- **TPL provides parallel *For* and *ForEach* loop that can make it easy to achieve data parallelism for arrays and collections.**

  For an example, see **PrimeCounter/Parallel**. This program finds prime numbers.

```
private static long[] FindPrimesParallel(
long first, int count)
{
   long lastExclusive = first + count;
   List<long> primes = new List<long>();
   if (first == 1)                  // 1 is not a prime
       first = 2;
   Parallel.For(first, lastExclusive, i =>
   {
       int numfact;
       Factors(i, out numfact);
       if (numfact == 1)
           primes.Add(i);
   });
   return primes.ToArray();
}
```

  You will implement this example as part of the lab.

Lab 7

Threading Techniques for Parallel Programming

In this lab you will use several different threading techniques to count prime numbers. Determining whether a large integer is a prime number is a compute-intensive operation, and performance improvements can be obtained with multiple-core CPUs by the use of parallel programming. You will compare several techniques.

Detailed instructions are contained in the Lab 7 write-up at the end of the chapter.

Suggested time: 60 minutes

Summary

- **You can use the *Thread* class to implement multithreading in .NET applications.**
- **You can use the *Monitor* class to program safe concurrent access to shared data.**
- **With the *ThreadPool* class you can obtain threads from a pool that is managed by the system.**
- **A foreground thread will keep the .NET execution environment running, while background threads will be stopped once all foreground threads have completed.**
- **There are various classes for synchronizing threads, including *ManualResetEvent, AutoResetEvent* and *CountdownEvent.***
- **You can use the Task Parallel Library to implement task parallelism and data parallelism in .NET applications.**

Lab 7

Threading Techniques for Parallel Programming

Introduction

In this lab you will use several different threading techniques to count prime numbers. Determining whether a large integer is a prime number is a compute-intensive operation, and performance improvements can be obtained with multiple-core CPUs by the use of parallel programming. You will compare several techniques.

Suggested Time: 60 minutes

Root Directory: **OIC\NetCs**

Directories:	**Labs\Lab7\PrimeCounter**	(do your work here)
	Chap07\PrimeCounter\Starter	(backup of starter code)
	Chap07\PrimeCounter\Threads	(answer to Part 1)
	Chap07\PrimeCounter\ThreadPool	(answer to Part 2)
	Chap07\PrimeCounter\Tasks	(answer to Part 3)
	Chap07\PrimeCounter\Parallel	(answer to Part 4)

Part 1. Using Threads

1. Open the starter project and examine the code. There is a class **Util** with a static method **CountPrimes()** that counts the number of primes in an interval beginning with **first**. This method relies on the method **FindPrimes()**, which returns an array of all the prime numbers in an interval. That method in turn relies on **Factors()**, which will factor a number. There is a test program that will enable you to interactively test these three methods. Build and run the program, satisfying yourself that they work.

2. Modify the test program to test only **CountPrimes()** with a hardcoded first number of one trillion and a count of 1000.

```
class Program
{
    const long BIGNUM = 1000000000000;      // one trillion
    const int COUNT = 1000;

    static void Main(string[] args)
    {
        Console.WriteLine("{0} total primes", Util.CountPrimes(BIGNUM,
            COUNT));
    }
}
```

3. Build and run without debugging. The result is 37 primes.

4. Add some instrumentation to your program to measure the time required for the computation. Use the **StopWatch** class from the **System.Diagnostics** namespace. Also, display the number of logical processors using the **ProcessorCount** property of the **Environment** class. Label the output "Sequential".

```
class Program
{
    const long BIGNUM = 1000000000000;      // one trillion
    const int COUNT = 1000;
    static Stopwatch stopWatch = new Stopwatch();

    static void Main(string[] args)
    {
        Console.WriteLine("Number Of Logical Processors: {0}",
            Environment.ProcessorCount);

        Console.WriteLine("Sequential");
        stopWatch.Restart();
        Console.WriteLine("{0} total primes", Util.CountPrimes(BIGNUM,
            COUNT));
        Console.WriteLine("elapsed time:\t{0}", stopWatch.Elapsed);
    }
}
```

5. Create a helper class **ThreadWithState** so that you will be able to pass **first** and count to the associated thread. Also maintain a static data member **TotalPrimes**. The thread procedure should count the primes in the interval and add this count to the total count. Both the count and total count should be displayed. Beware of a possible race condition in updating **TotalPrimes**. A simple solution is to use the **Interlocked.Add()** method. Finally display the elapsed time on the stopwatch. You will need to make **stopWatch** in the **Program** class public.

```
public class ThreadWithState
{
   public static int TotalPrimes = 0;
   public long First;
   public int Count;

   public void ThreadProc()
   {
      int numPrimes = Util.CountPrimes(First, Count);
      Console.WriteLine("{0} primes", numPrimes);
      // Add this threads count to the total
      Interlocked.Add(ref TotalPrimes, numPrimes);
      Console.WriteLine("{0} total primes", TotalPrimes);
      Console.WriteLine("elapsed time:\t{0}",
         Program.stopWatch.Elapsed);
   }
}
```

6. Finally, provide code in **Main()** to start two threads. The first thread will be for the first half of the interval beginning at BIGNUM, and the second thread for the second half of the interval. Use the **ThreadDemo** example as a model.

```
Console.WriteLine("Using Two Threads");
stopWatch.Restart();

ThreadWithState tws1 = new ThreadWithState { First = BIGNUM,
   Count = COUNT / 2 };
ThreadWithState tws2 = new ThreadWithState {
   First = BIGNUM + COUNT / 2, Count = COUNT - COUNT / 2 };
Thread t1 = new Thread(new ThreadStart(tws1.ThreadProc));
Thread t2 = new Thread(new ThreadStart(tws2.ThreadProc));
t1.Start();
t2.Start();
```

7. Build and run. Here is some sample output on a 2.8 GHz AMD processor with 6 cores and 8GB of system memory. This completes Part 1.

```
Number Of Logical Processors: 6
Sequential
37 total primes
elapsed time:   00:00:03.7012794
Using Two Threads
19 primes
19 total primes
elapsed time:   00:00:01.7747568
18 primes
37 total primes
elapsed time:   00:00:01.9316970
```

Part 2. Using the Thread Pool

1. In the previous solution we created individual threads and started them. Replace these four lines of code by two lines of code in which you call **QueueUserWorkItem()** from the **ThreadPool** class. Use the **WaitCallback** delegate.

```
ThreadPool.QueueUserWorkItem(new WaitCallback(tws1.ThreadProc));
ThreadPool.QueueUserWorkItem(new WaitCallback(tws2.ThreadProc));
```

2. There is a compile error. To use in the **WatiCallback** delegate you need to change the signature of the thread procedure.

```
public void ThreadProc(Object stateInfo)
```

3. Build and run. You don't get any output from the thread procedure for either thread! What is the difference between threads you create from the **Thread** class and threads obtained from the **ThreadPool** class?

4. Threads created from the **Thread** class by default are foreground threads, while threads from the **ThreadPool** class are background threads. Since background threads do not keep the managed execution environment running, once the main thread completes, the system will stop the background threads and shut down.

5. A quick and dirty way to keep the main thread from finishing before the threads you started from the thread pool is to have the main thread sleep for a few second:

```
Thread.Sleep(5000);
```

6. You can then build and run and see output from the thread procedure. A better approach is to have the main thread wait on a synchronization object. A convenient class to use in this context is **CountdownEvent**. You can initialize the counter to 2 (for the two tasks), and signal the event counter at the end of the thread procedure. We don't need to depend on the thread procedure for output any longer but can print the total number of primes and elapsed time in the main thread.

```
public class ThreadWithState
{
   public static int TotalPrimes = 0;
   public long First;
   public int Count;

   public static CountdownEvent cde = new CountdownEvent(2);

   public void ThreadProc(Object stateInfo)
   {
      int numPrimes = Util.CountPrimes(First, Count);
      Console.WriteLine("{0} primes", numPrimes);
      // Add this threads count to the total
      Interlocked.Add(ref TotalPrimes, numPrimes);
      cde.Signal();
   }
}

class Program
{
   ...
   static void Main(string[] args)
   {
      ...
      // Wait for threads to complete
      ThreadWithState.cde.Wait();

      Console.WriteLine("{0} total primes",
         ThreadWithState.TotalPrimes);
      Console.WriteLine("elapsed time:\t{0}", stopWatch.Elapsed);
   }
}
```

7. Build and run. This completes Part 2.

Part 3. Using Tasks

1. Modify the thread procedure to return the number of primes this thread has found as an integer. Also, the helper class does not need a synchronization object any longer, because the **Task<T>** class will cause an automatic wait until the result is available.

```
public class ThreadWithState
{
    public static int TotalPrimes = 0;
    public long First;
    public int Count;

    public int ThreadProc()
    {
        int numPrimes = Util.CountPrimes(First, Count);
        Console.WriteLine("{0} primes", numPrimes);
        // Add this thread's count to the total
        Interlocked.Add(ref TotalPrimes, numPrimes);
        return numPrimes;
    }
}
```

2. In **Main()** replace the two lines where you called **QueueUserWorkItem()** by code that instantiates **Task<int>** objects via the **Factory.StartNew()** method. In place of using a special delegate class, you can use lambda notation. The value returned by each thread can be accessed through the **Result** property.

```
Task<int> task1 = Task.Factory.StartNew(() => tws1.ThreadProc());
Task<int> task2 = Task.Factory.StartNew(() => tws2.ThreadProc());

// Task class causes automatic wait until results are available

Console.WriteLine("{0} total primes", task1.Result + task2.Result);
Console.WriteLine("elapsed time:\t{0}", stopWatch.Elapsed);
```

3. Import the **System.Threading.Tasks** namespace.

4. Build and run. This completes Part 3.

Part 4. Using Implicit Parallelism

A really powerful feature of the Task Parallel Library is its capability in many cases to implicitly cause parallel processing. Then the system can determine the optimal number of threads to use. The result is ease of programming and excellent performance.

1. Rather than manually starting threads in the main program, in our final solution we will modify the **Util** class to implement a parallel version of the **CountPrimes()** method, which in turn will rely on a parallel version of **FindPrimes()**. The heart of the computation is this **for** loop:

```
for (long i = first; i <= last; i++)
{
    int numfact;
```

```
    Factors(i, out numfact);
    if (numfact == 1)
        primes.Add(i);
}
```

2. We will replace it with **a Parallel.For** loop. Here is the complete code for the parallel version of our method. Again we use lambda notation, this time to specify the delegate method that will be invoked at each loop iteration.

```
private static long[] FindPrimesParallel(long first, int count)
{
   long lastExclusive = first + count;
   List<long> primes = new List<long>();
   if (first == 1)                    // 1 is not a prime
      first = 2;
   Parallel.For(first, lastExclusive, i =>
   {
      int numfact;
      Factors(i, out numfact);
      if (numfact == 1)
         primes.Add(i);
   });
   return primes.ToArray();
}
```

3. Import the **System.Threading.Tasks** namespace.

4. Implement the parallel version of **CountPrimes()**.

```
public static int CountPrimesParallel(long first, int count)
{
   return FindPrimesParallel(first, count).Length;
}
```

5. The main program now is very simple. There is not any thread code; we just call the two versions of **CountPrimes()** and display timing information using the stop watch.

```
class Program
{
   const long BIGNUM = 1000000000000;      // one trillion
   const int COUNT = 1000;
   static Stopwatch stopWatch = new Stopwatch();

   static void Main(string[] args)
   {
      Console.WriteLine("Number Of Logical Processors: {0}",
         Environment.ProcessorCount);

      Console.WriteLine("Sequential");
      stopWatch.Restart();
      Console.WriteLine("{0} total primes", Util.CountPrimes(BIGNUM,
         COUNT));
      Console.WriteLine("elapsed time:\t{0}", stopWatch.Elapsed);

      Console.WriteLine("Implilcitly Parallel");
```

```
      stopWatch.Restart();
      Console.WriteLine("{0} total primes",
         Util.CountPrimesParallel(BIGNUM, COUNT));
      Console.WriteLine("elapsed time:\t{0}", stopWatch.Elapsed);
   }
}
```

6. Build and run. Here is some sample output, using the same machine described earlier. The implicitly parallel version shows dramatic performance improvement, taking advantage of all six cores. This completes Part 4.

```
Number Of Logical Processors: 6
Sequential
37 total primes
elapsed time:   00:00:03.7762193
Implilcitly Parallel
37 total primes
elapsed time:   00:00:00.7838209
```