Exam Manual

LearnSmart

Microsoft .NET 2.0 Framework (70-536)

Application Developement Foundation

This LearnSmart exam manual will equip you with all the knowledge necessary to successfully complete the Microsoft .NET 2.0 Development exam (70-536). By studying this manual, you will become familiar with an array of exam-related content, including:

- Developing applications that use system types and collections
- Implementing service processes, threating and application domains in a .NET Framework application
- And more!

Give yourself the competitive edge necessary to further your career as an IT professional and purchase this exam manual today!

Microsoft .NET 2.0 Application Development (70-536) LearnSmart Exam Manual

Product ID: 10725
Production Date: July 22, 2011

Warning and Disclaimer

Every effort has been made to make this document as complete and as accurate as possible, but no warranty or fitness is implied. The publisher and authors assume no responsibility for errors or omissions. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this document.

Volume, Corporate, and Educational Sales

PrepLogic offers favorable discounts on all products when ordered in quantity. For more information, please contact PrepLogic directly:

1-800-418-6789
solutions@learnsmartsystems.com

International Contact Information

International: +1 (813) 769-0920

United Kingdom: (0) 20 8816 8036

Table of Contents

Abstract

This Exam Manual will help prepare students to pass the certification exam for 70-536: TS: Microsoft .NET Framework 2.0—Application Development Foundation. This exam is one of the examinations required for the following three Microsoft Certified Technology Specialist (MCTS) certifications:

- Microsoft Certified Technology Specialist: .NET Framework 2.0 Web Applications
- Microsoft Certified Technology Specialist: .NET Framework 2.0 Windows Applications
- Microsoft Certified Technology Specialist: .NET Framework 2.0 Distributed Applications

Microsoft Technology Specialists are capable of building, implementing, troubleshooting, and debugging applications using Microsoft technologies. Each of the above certifications requires that a candidate pass the 70-536 exam and one additional exam specific to each certification path

When taking the 70-536 exam, candidates may select the programming language in which all code examples will appear. When the exam begins, they must select one of the following languages:

- Microsoft Visual Basic 2005
- Microsoft Visual C# 2005
- Microsoft Visual C++ 2005

The 70-536 exam consists of 150 questions designed to measure candidates' knowledge and competency in the fundamentals of the.NET Framework 2.0. This exam covers new features and enhancements in the following areas, specifically, the enhanced capabilities of the namespaces. The following topics are covered in this exam:

- **Fundamentals of the .NET Framework and an overview of new features, enhancements, and capabilities of the .NET Framework 2.0**
- **Service processes, threading, and application domains (41 questions)**
- **Configuration, diagnostic, management, and installation features (21 questions)**
- **Serialization and input/output functionality (26 Questions)**
- **The .NET Framework 2.0 security features (29 Questions)**
- **Interoperability, reflection, and mailing functionality (16 Questions)**
- **Globalization, drawing, and text manipulation functionality (17 Questions)**

What to Know

The best way to prepare for this exam is to read all the material you can find related the .NET and .NET Framework 2.0. Also, you should try to get as much hands on experience with the .NET framework as possible. This is accomplished in three ways: by Coding, Reading, and Studying. Microsoft provides a wonderful web resource called the MSDN library that contains sample code, documentation, and other materials that you can use to further your knowledge of the .NET Framework.

It's a good idea to also participate in several large projects. Whether you do these projects at your office or you simulate them on your own, it's wise to involve yourself in the practical coding of large scale multi-threaded .NET intensive apps that use a wide variety of new extensions to the .NET Framework you will find in this Exam Manual. Also, you should check back at www.PrepLogic.com to see the newly upcoming leading practice exam, which will be available for an extremely low price.

Lastly, it's highly recommended that you familiarized yourself with the objectives of the exam at:
http://www.microsoft.com/learning/exams/70-536.asp

Tips

This exam requires a thorough understanding of the .NET Framework as well as the new capabilities specifically provided by the .NET 2.0 Framework. Depending on the programming language you select, you will need to be familiar with the Visual Basic 2005, Visual C# 2005, or Visual C++ 2005. You will not be required to write and compile code, but you should be familiar with syntax and structure of the language selected.

The .NET Framework 2.0 does not introduce a new development framework or paradigm, implementation model, or scripting model from earlier releases of .NET Framework 1.0 or .NET Framework 1.1. However, because it is much more than an incremental release, it introduces significant enhancements to the .NET Framework. This manual does not cover each technical item in detail, but it provides a targeted overview of the material in this exam and prepares you to understand the topics outlined in the Technology Specialist (TS) Exam 70-536: TS: Microsoft .NET Framework 2.0—Application Development Foundation exam.

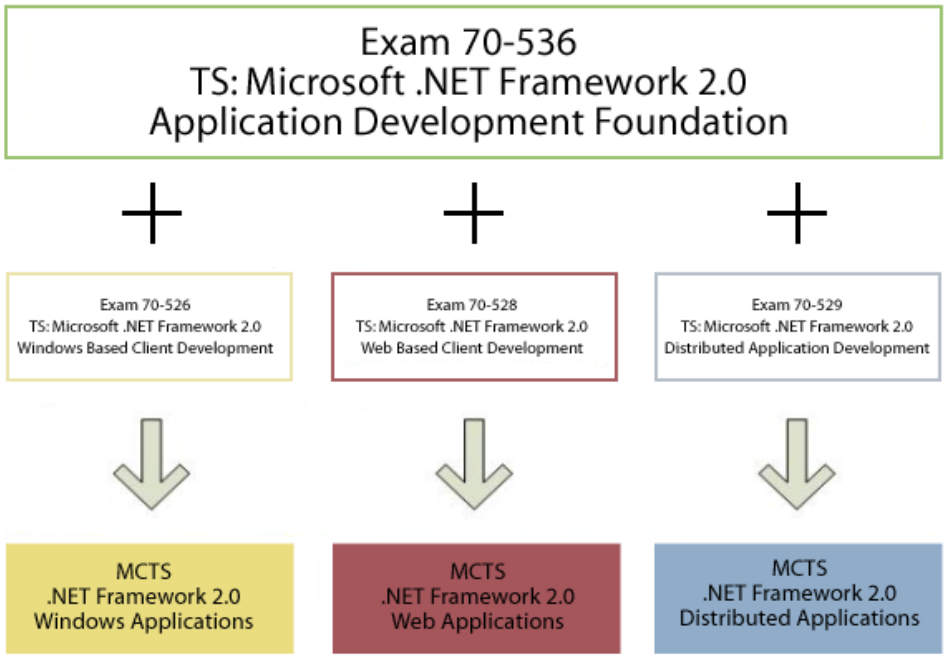
Overview of new features, enhancements, and capabilities of the .NET Framework 2.0

1.1 Microsoft .NET Framework 2.0 Certification Path

This Exam Manual will help prepare you for the certification exam for 70-536: TS: Microsoft .NET Framework 2.0—Application Development Foundation. This exam is one of the examinations required for one of the following three Microsoft Certified Technology Specialist (MCTS) certifications:

- Microsoft Certified Technology Specialist: .NET Framework 2.0 Web Applications
- Microsoft Certified Technology Specialist: .NET Framework 2.0 Windows Applications
- Microsoft Certified Technology Specialist: .NET Framework 2.0 Distributed Applications

Microsoft Technology Specialists are capable of building, implementing, troubleshooting, and debugging applications using Microsoft technologies. The above certifications require that a candidate pass the 70-536 exam and one additional exam specific to each certification path. The certification path for all three options is follows:



MCTS Certification Path Options

Candidates for the 70-536 exam can select the programming language in which all code examples will appear. When the exam begins, they must select from one of the following:

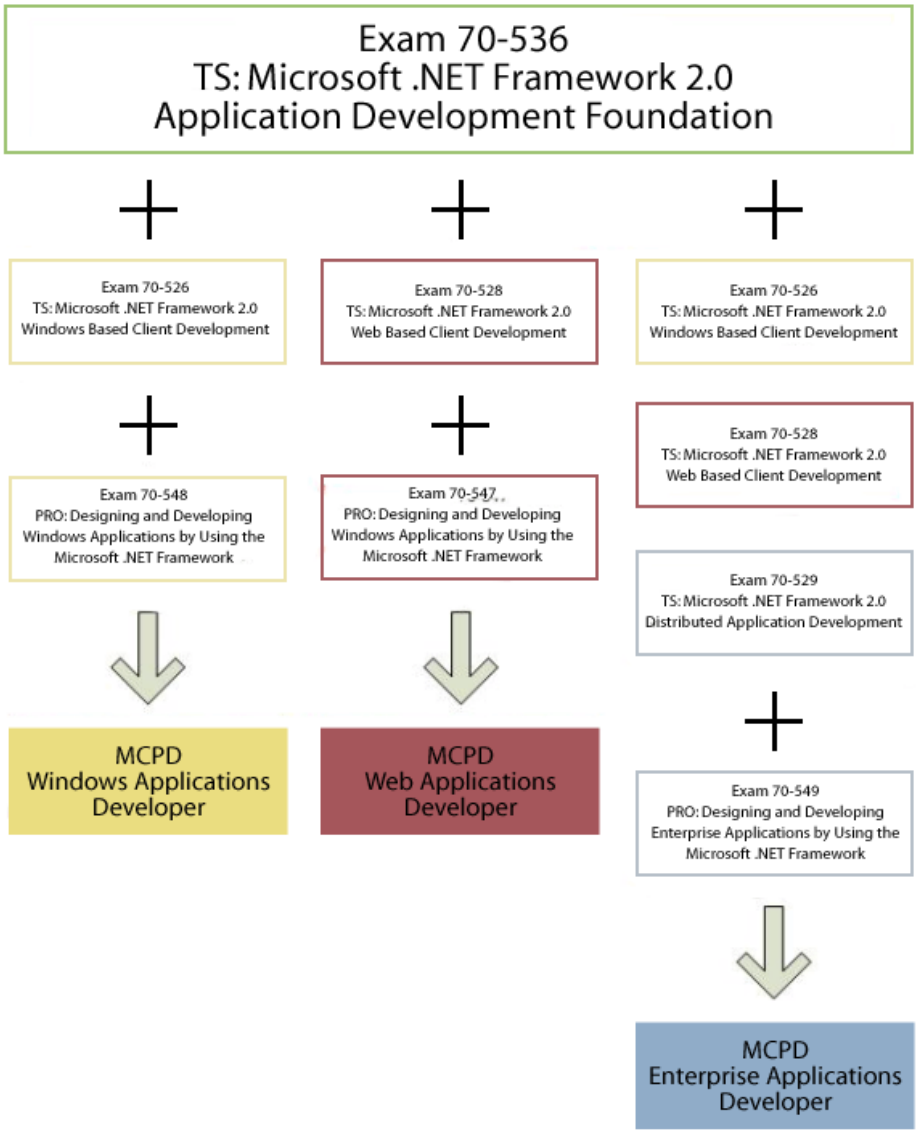
- Microsoft Visual Basic 2005
- Microsoft Visual C# 2005
- Microsoft Visual C++ 2005

The 70-536 exam consists of 150 questions that measure candidates' knowledge of the fundamentals of the.NET Framework 2.0. It covers new features and enhancements in the following areas, specifically, the enhanced capabilities of the namespaces.

After achieving a Microsoft Certified Technology Specialist (MCTS) certification, candidates can enhance their certification level by becoming a Microsoft Certified Professional Developer (MCPD). The MCPD certification is targeted to a candidate's specific technical expertise. The three MCPD certifications are:

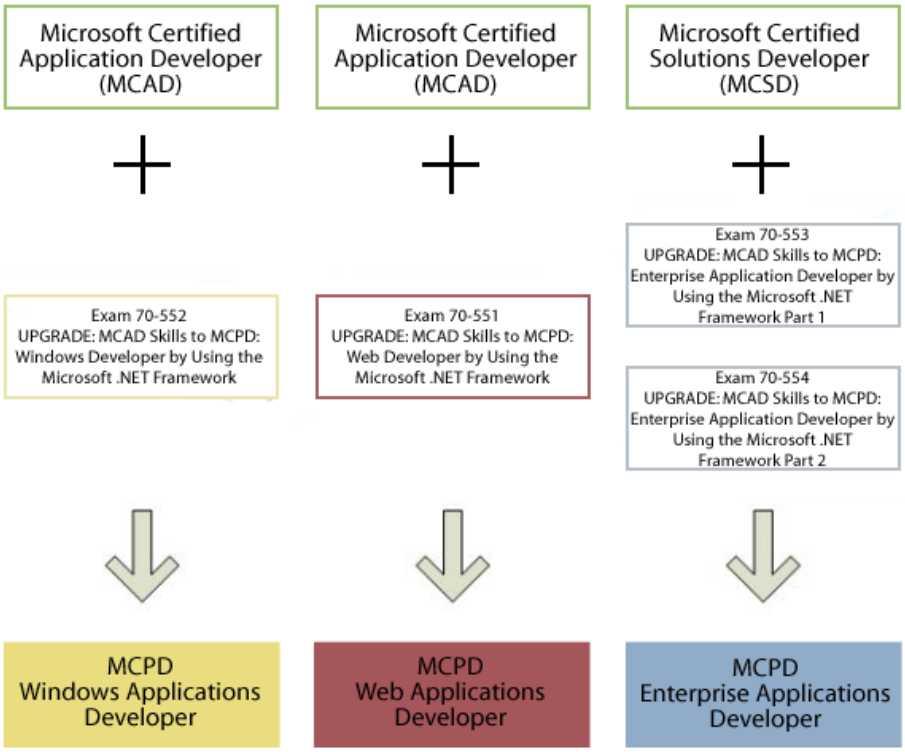
- Microsoft Certified Professional Developer: Web Applications Developer
- Microsoft Certified Professional Developer: Windows Applications Developer
- Microsoft Certified Professional Developer: Enterprise Applications Developer

Each of the above certifications requires a candidate to pass one additional exam specific to a certification path (the MCPD: Enterprise Applications Developer requires all three MCTS exams in addition to the MCPD exam). The certification path for all three options is displayed below:



MCPD Certification Path Options

Candidates who have achieved certification as either a Microsoft Certified Application Developer (MCAD) or Microsoft Certified Solutions Developer (MCSD) can upgrade to the MCPD level by taking the following upgrade exams:



Upgrade Certification Path Options

1.2 Fundamentals of the .NET Framework

This manual assumes that you are familiar with the .NET Framework and have been developing applications (Windows Forms/Windows GUI Applications, ASP.NET Web applications, Mobile applications, Windows Services, Console Applications, XML Web Services, etc.). The Exam 70-536: TS: Microsoft .NET Framework 2.0—Application Development Foundation primarily covers enhancements made to the .NET Framework, specific to version 2.0, but we will briefly review the .NET Framework. For more detailed information on the .NET Framework and information on each of the examination options available, refer to the Introduction.

The .NET Framework is a language-neutral component library and execution environment that provides the infrastructure for building applications using .NET. Application developers can build powerful, enterprise, integrated applications regardless of platform or language. The .NET Framework 2.0 does not introduce a new development framework or paradigm, implementation model, or scripting model from the previous release of .NET Framework 1.0 or .NET Framework 1.1 (the .NET Framework Release 1.0 was initially released in January of 2002). But because it is more than an incremental release, it introduces some significant enhancements to the .NET Framework.

The .NET framework is an object-oriented hierarchy of classes that are ubiquitous to all Microsoft Windows operating systems, regardless of whether the user is developing applications for Windows Forms, Web applications, mobile applications, services, or any other task. When creating a .NET application, a developer creates

a class, and defines the class properties, events, and methods that build the functionality of the application. With .NET, these classes support object-oriented features such as polymorphism, inheritance and encapsulation. For example, when executing an ASP.NET application, the ASP.NET runtime engine will transform the source code the .aspx page into an instance of a .NET framework class that inherits from the *Page* base class. This fundamental concept is why this exam (Exam 70-536: TS: Microsoft .NET Framework 2.0—Application Development Foundation) must be passed in tandem with any of three other exams to earn a Microsoft Certified Technology Specialist certification (with a focus on either.NET Framework 2.0 Web Applications, .NET Framework 2.0 Windows Applications, or .NET Framework 2.0 Distributed Applications).
The .NET Framework has two main components:

- The common language runtime (CLR)
- The .NET Framework class library

The common language runtime (CLR) is at the foundation of the .NET Framework. The CLR provides a run-time environment and run-time services for .NET applications and is responsible for code management, access security, language interpretation, memory management, thread management, process management, compilation and code accuracy, strict type and code verification (the common type system of CTS) to ensure that all managed code is self-describing. Code management is an important principle in the .NET Framework. Code targeting the runtime is managed code (custom object libraries, class libraries), while code not targeting the runtime is unmanaged code (Internet Information Services, ASP.NET web applications). The CLR also provides a management environment that automatically provides common services (garbage collection and security). In addition, ASP.NET provides the environment to incorporate both managed and unmanaged features, such as an unmanaged component or application such as Internet Explorer with an embedded managed component or Windows Forms control.

As an application developer or system administrator, you are concerned with productivity, reliability and performance. Because the runtime automatically handles object layout as well as managed references to objects in memory, they are automatically released when no longer being used. This feature helps eliminate memory leaks and invalid object memory references. Finally, the Just In Time complier (JIT) allows for managed code to run in the system's native machine language. Also, because the CLR supports development in multiple development languages, application developers can script in their language of choice while still accessing a common runtime and class library. Finally, the CLR also provides other fundamental services which are outside the scope of this book.

However, this book will focus on the .NET Framework class library, specifically, on the features, enhancements, and capabilities of the class libraries available in the .NET Framework 2.0. Microsoft defines the .NET Framework as:

"…a comprehensive, object-oriented collection of reusable types that you can use to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services."

In short, the .NET Framework contains the object-oriented, hierarchical, extensible classes, interfaces, and value types that provide the framework to build .NET applications, components and controls. They provide the standard functionality that allows developers to utilize string manipulation, security management, network communications, thread management, input/output controls and user interface design features. The .NET Framework provides class libraries (APIs) that replace the libraries and frameworks previously used by application developers based upon their programming language (Microsoft Foundation Classes for C++ developers, Visual Basic runtime for VB developers, etc.). The .NET Framework simplifies those issues by providing a common set of APIs for all the supported programming languages (at the time of this writing, more than 25 languages are supported). The .NET Framework supports cross-language inheritance (you can inherit classes across language boundaries), error handling and debugging. A class written in one

language is reusable by classes written in other languages.
Note: The European Computer Manufacturers Association (ECMA) standard called the Common Language Infrastructure (CLI) defines the Common Language Specification (CLS) rules for language interoperability. Code written in a CLS-compliant language is interoperable with code written in another CLS-compliant language.

The .NET Framework currently has four CLS-compliant languages:

- Microsoft Visual Basic .NET
- Microsoft Visual C#
- Microsoft Visual C++ .NET
- Microsoft Visual J# .NET

The compiler generates the code as Microsoft Intermediate Language (MSIL) which makes the programs (written in one of the languages specified above) CLS compliant and thus, interoperable with application written in other CLS-compliant languages. The CLR loads the MSIL and executes the code when the application is run.

The .NET Framework also allows multiple versions of applications on the same system by using assemblies, which are deployment units in the .NET Framework. This capability is sometimes called side-by-side execution. The assembly contains the Microsoft Intermediate Language (MSIL) code and metadata with information about the name and version of the assembly and other assemblies on which the current assembly depends. This structure allows the CLR to use the name and version information in the metadata to run multiple versions of an application side by side.

As published by Microsoft, the .NET Framework was designed to manage the "plumbing" typically inherent to application development, allowing developers to focus on business logic. The.NET Framework's goals are to:

- Make it easy to build, deploy and administer secure, robust and high-performing applications.
- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and version conflicts.
- To provide a code-execution environment that promotes safe execution of code (including code created by an unknown or semi-trusted third party).
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications (such as Windows-based applications and Web-based applications).
- To build all communication on industry standards so that code based on the .NET Framework can integrate with any other code

Using the .NET Framework, developers can efficiently create the following types of applications and services:

- Windows Forms/Windows GUI applications
- ASP.NET Web applications
- Windows services
- Console applications
- XML Web Services

The .NET Framework class library is tightly integrated with the CLR and provides a class library. It also implements interfaces for developers to create custom classes that integrate seamlessly with the existing classes of the .NET Framework while providing class libraries to accomplish very specialized (and powerful) tasks. .NET Framework 2.0 expands on those specialized classes considerably, as outlined within this book.

1.3 New features, enhancements, and capabilities of the .NET Framework 2.0

The .NET Framework version 2.0 provides significant enhancements to its predecessor (version 1.1). A summary of some of those enhancements is provided here. Detailed enhancements (relevant to the exam material) will be provided in this book, especially enhancements made to various namespaces.

Note: Namespaces are organized hierarchically, contain logically related and reusable classes, and divide an assembly into a logical grouping of types. Multiple assemblies can use the same namespace.

Links to resources regarding other enhancements to the .NET Framework 2.0, while important, are outside the scope of this book. They will be provided at the end of this chapter for further reference.

- New classes to create and modify the Access Control List (ACL). These enhancements are discussed in Chapter 6 – Examination of the .NET Framework 2.0 Security Features.
- Support for new Authentication stream classes to secure information transmitted between the client and the server (mutual authentication, data encryption, data signing and support for the Secure Sockets Layer (SSL). These enhancements are discussed in Chapter 6 – Examination of the .NET Framework 2.0 Security Features.
- New features in ADO.NET:
    - Support for user-defined type (UDT)
    - Support for asynchronous database operations
    - Support for XLM data types
    - Support for Large value types
    - Support for snapshot isolation
    - Support for multiple active result sets (MARS) with SQL Server 2005

- New ASP.NET features:
  - New controls
  - New caching features
  - New support for Web parts
  - New support for master pages
  - Support to deploy Web applications to production servers without source code
  - New device filtering to render output to different devices and browsers
- Improved support for interoperability with COM. These enhancements are discussed in Chapter 7 – Examination of Interoperability, Reflection, and Mailing Functionality:
  - New support for manipulating operating system handles
  - Improvements to marshaling
  - Improvements in performance with calls between applications in different application domains
  - Elimination of the dependency on the registry to resolve type library references with the addition of new switches on the Type Library Importer and Type Library Exporter
- New support by the NetworkChange class to allow applications to receive notification when the Internet Protocol (IP) address of the network interface (network card or network adapter) changes
- Enhancements to the System.NET namespace to provide additional support for distributed computing:
  - Support for FTP client requests
  - Support for caching of HTTP resources
  - Support for automatic proxy discovery
  - Ability to obtain network traffic and statistical information
  - Addition of HTTPListener class to allow the creation of simple Web servers for responding to HTTP requests
  - Security and performance enhancements have been made to the System.Net.Sockets and System.Uri classes
  - Support for SOAP 1.2 and nullable elements in the System.Runtime.Remoting.Channels namespaces
  - Support for authentication, encryption and additional security enhancements to the System.Runtime.Remoting.Channels namespace

- Ability to use custom DLLs for EventLog messages, parameters and categories These enhancements are discussed in Chapter 4 – Examination of configuration, diagnostic, management, and installation features.
- Added support for X.509 certificate stores, chains and extensions. Also added ability to sign and verify XML using X.509 certificates without using platform invoke. Also new support for PKCS7 signature and encryption and CMS.
- New access to File Transfer Protocol (FTP) resources via the WebRequest, WebResponse and WebClient classes.
- Using the System.Net.Cache namespace, applications can control the caching of the WebRequest, WebResponse, and WebClient classes.
- Addition of new namespaces System Namespace and System.Collections.Generic that provide the ability to create generic classes, structures, interfaces, methods and delegates to be declared and defined with unspecified (or generic) type parameters instead of specific types. Generics are currently supported in Visual Basic, C# and C++. Enhancements have been made to the System.Type and System.Reflection.MethodInfo to allow the examination and manipulation of generic types. These enhancements are discussed in Chapter 2 – Examination of System Types and Collections.
- Additional support for globalization features that support developing applications for different languages/cultures. These enhancements are discussed in Chapter 8 – Examination of globalization, drawing, and text manipulation functionality.
    - Ability to define and deploy culture-related information as needed.
    - Improved support for Unicode character mapping, encoding and support for the latest normalization standard defined by the Unicode consortium.
    - Improvements to the GetCultureInfo method of the System.Globalization. CultureInfo namespace.
- Enhancements to the I/O classes to improve reading and writing text files and obtaining drive information. These enhancements are discussed in Chapter 5 – Examination of serialization and input/output functionality.
- Improvements in the Remoting capability to now support IPv6addresses, the exchange of generic types, and enhancements to the System.Runtime.Remoting.Channels.Tcp and System. Runtime.Remoting.Channels.Ipc namepaces.
- Addition of new members to the Console class.
- Support for 64-bit platforms.
- New Data Application API (DPDAI) allowing applications to encrypt passwords, keys and connection strings without needing to call platform invoke and the ability to encrypt blocks of memory (Windows Server 2003 or later).
- Ability to control the Debugger display.
- Debugger enhancements that allow developers to change source code while debugging an application in Break mode, then continue and resume the application to observe the effect in any programming language supported by Visual Studio.

- Enhancements to the System.Net.NetworkInformation namespaces to allow access to IP, IPv4, IPv6, TCP and UDPnetwork traffic statistics. Applications can now view address and configuration information for the local computer's network adapters.
- The new Ping class provides the ability to determine if a remote computer is accessible over the network (supporting asynchronous and synchronous calls).
- Significant enhancements have been made to the following programming languages (for more information, see the URL references located at the end of this chapter):
    - C# 2.0
    - Visual J#
    - Microsoft C/C++
    - Visual Basic
- Enhancements to the System.Security.SecurityException class now provide additional data regarding security exceptions. Enhancements to the System.Security namespace are discussed in Chapter 6 - Examination of the .NET Framework 2.0 security features.
- A new System.IO.Ports.SerialPort class allows applications to access the computer serial ports and communicate with the serial I/O devices.
- Serialization capabilities have been improved with the BinaryFormatter and SoapFormatter classes to support "version-tolerant serialization," which allows a type to be deserialized from the serialization of a different version. These enhancements are discussed in Chapter 5 - Examination of serialization and input/output functionality.
- XML serialization now has properties instead of fields to represent schema elements. It also has been modified to support the serialization of generic types (see Chapter 2 - Examination of system types and collections), the use of the Nullable structure. These enhancements are discussed in Chapter 5 - Examination of serialization and input/output functionality.
- Improvements to the System.Net.Mail and System.Net.Mime namespaces allow applications to send e-mail to one (or more) recipients, include attachments, and send carbon and blind carbon copies. These enhancements are discussed in Chapter 7 - Examination of interoperability, reflection, and mailing functionality.
- Strongly Typed Resource Support so The Resource File Generator creates resource files embedded in executable files/satellite assemblies. The Resource File Generator produces a wrapper class for each resource file, giving the developer access to resources and preventing spelling mistakes in resource names.
- New classes that support tracing and logging of system events related to I/O, and application startup and shutdown. Support for trace data filtering allows developers to specify the type of data to log.
- The addition of the Systems.Tranasctions namespace allows developers to have applications use the local transaction manager or Microsoft Distributed Transaction Coordinator (MSDTC).
- Web Services now support SOAP 1.2 and WS-I Basic Profile 1.0. Developers can invoke Web methods asynchronously using an event-based programming pattern.

- Significant enhancements have been made to the System.XML namespace. Creating System.XML.XmlReader and System.XML.XMLWriter has been modified; they now include Type support and perform better. The 2.0 Framework provides a new XSL Transformation processor (XSLT) and enhancements to the System.XML.XPath.XPathNavigator namespace. These enhancements are explained in Chapter 5 - Examination of serialization and input/output functionality.

Specific enhancements to the .NET Framework version 2.0 that are significant to a specific certification (MCTS: .NET Framework 2.0 Web Applications, MCTS: .NET Framework 2.0 Windows Applications, MCTS: .NET Framework 2.0 Distributed Applications) are not discussed here. For example, enhancements to Web Service or Visual Basic are outside the scope of this exam. Other enhancements, while significant, are outside the scope of the 70-536:TS: Microsoft .NET Framework 2.0—Application Development Foundation exam, and thus, outside the scope of this book.

You can use the following resources to locate additional information:

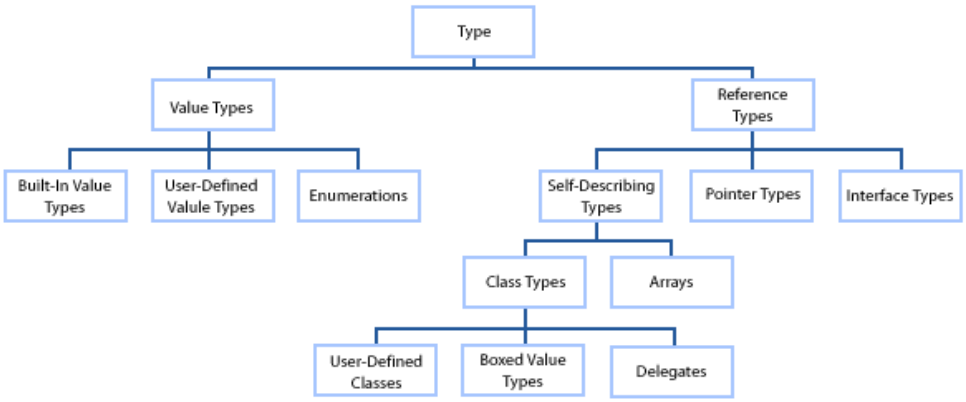
- MSDN - What's New in the .NET Framework 2.0 - http://msdn2.microsoft.com/en-us/library/t357fb32(VS.80).aspx
- MSDN - .NET Framework Developer Center - http://msdn.microsoft.com/netframework/
- MCTS 70-536 Exam Prep : Microsoft .NET Framework 2.0 Foundation Exam by Amit Kalani
- MCTS 70-528 Exam Prep : Microsoft .NET Framework 2.0 Web-based Client Development Exam (Exam Prep) by Amit Kalani
- MCTS Self-Paced Training Kit (Exam 70-536): Microsoft .NET Framework 2.0 Application Development Foundation (Pro – Developer)

Developing applications that use system types and collections

2.1. Defining system types and collections

In the .NET Framework, the class object is the root of the inheritance hierarchy in the .NET Framework. Therefore, every class in the .NET Framework derives from this class. Consequently, if you define a class without specifying any other inheritance, the Object class is the implied base class. The Object class provides the basic properties and methods that all objects must support (for example, returning an identifying string, returning a Type object, etc.). All types are derived from the *System.Object* namespace.

The .NET Framework provides two kinds of types, value and reference. Value types, the simplest types in the .NET Framework, are represented by a series of bits that derive their class from the base class *ValueType*. Reference types are represented as a location for a sequence of bits. The instance of the reference type is allocated from the managed common language runtime heap and a reference to an object is returned. Both types are explained in more detail later in this chapter. System types, value types, reference types and their derived classes are displayed below.



The *System.Collections* namespace contains interfaces and classes which define various "collections" of objects. These "collections" objects are lists, arrays, queues, hash tables and dictionaries. They allow developers to group related objects as well as iterate, look up, store, sort and manage collections of those objects. This class is described in more detail later in this chapter.

2.2. Managing data in a .NET Framework application by using system types

System Namespace

The System Namespace is the root of all the namespaces in the .NET Framework. It contains all the other namespaces, and the fundamental .NET classes and base classes that define the commonly used value and reference data types, events, event handlers, interfaces, attributes and processing exceptions.

Value Types

Value types are typically small variables, represented by a series of bits and accessed primarily for a single data value (numeric and Boolean types). They are still objects (and have properties and methods associated with them) but they contain their data directly, stored in the memory stack, rather than referencing data stored elsewhere. Value Types derive their class from the base class *ValueType* (*System.Value* base type). The three value types are:

- Built-in value types – Base types in the .NET Framework from which other types are built. All built-in numeric data types are value types.
- User-defined value types – Also called Structure (or Structs); similar to classes. As value types, instances user-defined types are stored on the stack. They are composites of other types that make it easier to work with related data.
- Enumerations – Related symbols (lists) that have fixed values that improve code readability and simplify coding since you can assign meaningful symbols to values rather than using numeric values.

The following are some of the more common value types are available in the .NET Framework (more than 300 value types are available):

Value Type	CLS Compliant*	Description
System.Boolean	Yes	Boolean value (true or false) (4 Bytes)
System.Byte	Yes	8-bit unsigned integer (1 Byte)
System.Char	Yes	UTF-16 code point (2 Bytes) (4 Bytes)
System.DateTime	Yes	An instant in time, typically expressed as a date and time of day (8 Bytes)
System.Decimal	Yes	Decimal number (16 Bytes)
System.Double	Yes	Double-precision floating-point number (8 Bytes)
System.Enum		Base class for enumerations
System.Int16	Yes	16-bit signed integer (2 Bytes)
System.Int32	Yes	32-bit signed integer (4 Bytes)
System.Int64	Yes	64-bit signed integer (8 Bytes)
System.IntPtr	Yes	Signed Integer that is platform independent
System.Object	Yes	Root object
System.SByte	No	8-bit signed integer (1 Byte)
System.Single	Yes	Single-precision floating-point number (4 Bytes)
System.String	Yes	Fixed-length string of Unicode characters
System.TimeSpan	Yes	Time interval
System.UInt16	No	16-bit unsigned integer (2 Bytes)
System.UInt32	No	32-bit unsigned integer (4 Bytes)
System.UInt64	No	64-bit unsigned integer (8 Bytes)
System.UIntPtr	No	Unsigned Integer that is platform independent

*__Note:__ The European Computer Manufacturers Association (ECMA) standard called the Common Language Infrastructure (CLI) defines the Common Language Specification (CLS) rules for language interoperability. Code written in a CLS-compliant language is interoperable with code written in another CLS-compliant language. Therefore, programmers should avoid using signed Byte or unsigned Integer values to achieve cross-language interoperability.

All types ending in *16* are referred to as *WORD* size values in the WIN32 platform. All types ending in *32* are referred to as *DWORD* size values in the WIN32 platform. All types ending in *64* are referred to as *QWORD* size values in the WIN32 platform.

Programmers usually do not use the full system value type name when using value types. For example, programmers often use *Int* rather than *System.Int32* in their code. Value types have an implicit constructor. Therefore, declaring a value type instantiates it automatically so the *New* keyword need not be included as it does with other classes. While the constructor automatically assigns a default value to the value type (0 or null) to the new instance, you should always explicitly initialize the variable as shown below:

VB

```
Dim myBoolean as Boolean = False
```

or

C#

```
bool myboolean = false;
```

Nullable type

The Nullable class is new to the .NET Framework 2.0. It supports a value type that can be assigned a null reference like a reference type (in Visual Basic, this is *Nothing*). Therefore, you can determine whether the value has or has not been assigned. The class cannot be inherited. A type is considered nullable when it can be assigned a value or assigned a null reference. When a type is assigned a null reference, the type has no value. Some reference types are nullable while others are not. For example, the *System.String* reference type is nullable because it can contain an actual value or be expressed as a value of null. A value type such as Int64 cannot be nullable because it can be expressed only as a value of its type and null is not a value Int64 value.

The public constructor *Nullable* Initializes a new instance of the Nullable structure to the specified value. Two public properties exposed by the Nullable generic type are:

- HasValue - Gets a value indicating if the current Nullable object has a value (returns true if the current Nullable object has a value, False if it has no value)
- Value - Gets the actual value of the Nullable value (if the HasValue property is true). If there is no value of the Nullable object, an "InvalidOperationException" exception will be thrown.

Some of the public methods exposed by the Nullable generic type are:

- Equals - Indicates if a Nullable value is equal to another object. This method is Overloaded, therefore the following signatures are valid:
    - Nullable.Equals (Object)

Indicates if the specified Object is equal to the current Nullable object.

- Nullable.Equals (Object, Object)

Indicates if the specified Object instances are considered equal to one another. Returns true if the objects are equal to one another, are both null references, or are actually the same object instance.

The Equals method supports reference equality for reference types (indicating that the object references being compared are referring to the same object) and supports bitwise equality for value types (indicating that the objects have the same binary representation).

- GetValueOrDefault - Returns the value of the current Nullable object or a default value determined by the signature used. This method is Overloaded, therefore the following signatures are valid:
    - Nullable.GetValueOrDefault() - Returns the value of the current Nullable object or the object's default value
    - Nullable.GetValueOrDefault(defaultVal) - Returns the value of the current Nullable object or the specified default value
- op_Expicit - Returns the value (of the Value property) of the Nullable value. A static method.
- op_Implicit - Creates a new nullable object initialized to specified value. A static method.
- ReferenceEquals - Determines if the Object instances are the same instance. Returns true if the objects are referencing the same instance or if both objects are null references. A static method.
- ToString - Returns the text representation of the value of the Nullable object if the HasValue property is true. If the HasValue property is false, returns an empty string (""). The string value returned by this method is the same value returned by the ToString method of the object returned by this object's Value property.

The Nullable type supports the Nullable generic structure (which is new to the .NET Framework 2.0). The Nullable generic structure represents an object whose underlying type is a value type that can be assigned a null reference.

The Nullable type supports obtaining the underlying type of the Nullable type as well as comparison and equality operations of Nullable types whose underlying value types do not support generic comparison and equality operations. For example, a field value may or may not have an actual value depending on the application's circumstances. If you define the field as a Nullable type rather than a value type, it can support the value that it does not exist (e.g., *Nothing* when using Visual Basic).

If a Nullable type is boxed, the common language runtime automatically boxes the underlying value of the Nullable object, not the Nullable object itself (boxing and unboxing are explained later in this chapter). As example of using the Nullable type is as follows:

VB

```
Dim myBoolean as Nullable(Of Boolean) = Nothing
```

or

C#

```
Nullable<bool>myboolean = null;
```

Reference Types

All objects that are not value types (derived from the (derived from the *System.ValueType* namespace) are reference types. About 2500 built-in reference types are available in the .NET Framework. Reference types are types that are represented as a location for a sequence of bits. When a reference object is created, the instance of the reference type is allocated from the managed common language runtime heap and a reference to the object is returned (a pointer to the object stored on the stack). For example, the String object is a reference type (not a value type) that contains an immutable series of one or more characters. Direct access to the object is not allowed so that the garbage collector can track the any references to the object and release the data once the references are released.

Attributes

Attributes hold read-only information that programmers place in an object's metadata. The attributes describe the object. Attributes can be provided by the .NET Framework or defined and used programmatically within your code. All attributes are derived from the base class *System.Attribute*. Attributes describe types, properties and methods so they can return using reflection. The information stored in the attributes is not only descriptive data; it also can declare requirements for an assembly (such as adding the *Serializable* attribute to enable a class to be serialized).

For example, *System* Namespace attributes are displayed below:

System Namespace Attributes

Name	Description
AttributeUsageAttribute	Specifies the target types that other attribute classes can be applied (assembly, class, method, etc.)
CLSCompliantAttribute	Indicates if the application is compliant with the Common Language Specification (this is described in the Chapter 1)
FlagsAttribute	Indicates if the enumeration can be treated as a bit field (flag)
ObsoleteAttribute	Used to indicate which program elements are obsolete and no longer usable by the application

Generic Types

Generics are a new feature of the .NET Framework 2.0 (and are emphasized in this exam). Generics allow programmers to create generic type parameters (as opposed to specific types) of classes, structures, interfaces, methods and delegates. Thus, you can define a type but leave the details of the class unspecified so it can be programmatically specified in the code. Generics are declared and defined with unspecified, generic type parameters. Once the generic object is used, the actual type is specified. Generic classes are supported by the System Namespace and *System.Collections.Generic* namespace. These classes are *Dictionary, Queue, SortedDictionary,* and *SortedList*.

*__Note:__ Generics are supported in Visual Basic, C#, and C++

New members have been added to the *System.Type* and *System.Reflection.MethodInfo* namespaces to identify generic types. Using reflection, programmers can examine and manipulate generic types and methods during runtime. Before Generic classes, programmers were forced to use the Object class to achieve the same functionality. But Generics are type-safe, can reduce run-time errors, and increase performance, making them the preferred choice.

Exception Classes

Exception classes represent errors thrown during the application execution . The *System.Exception* class is the base class for all exceptions.

When an error is encountered, the system or currently executing application reports the error by throwing an exception, which contains information about the error. The exception is handled by the default exception handler or the application (if programmed to do so). The CLR allows programmers to use **try** and **catch** blocks to catch and handle exceptions/errors (which are represented by exception objects). The *Message* property provides a friendly description of the error and a suggested resolution. The *StackTrace* property provides an attack trace with more detailed information and information on where the exception occurred (listing the called methods, source file and line number where the calls are made).
The two types of categories under the base class Exception are:

- SystemException – pre-defined common language runtime exceptions
- ApplicationException – user-defined application exception classes

Some of the public properties exposed by *System.Exception* namespace are:

- Data – Gets a collection of key/value pairs providing additional user-defined information about the exception
- HelpLnk - Gets or sets a link to the help file associated with the exception
- InnerExcpetion - Gets the Exception instance that caused the exception
- Message - Gets a message that describes the exception
- Source - Gets or sets the name of the application or the object that causes the exception
- SrackTrace - Gets a string representation of the frames on the call stack at the time the exception was thrown
- TargetSite - Gets the method that throws the exception

The protected property exposed by *System.Exception* namespace is:

- HResult – Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception

Some of the public methods exposed by *System.Exception* namespace are:

- Equals - Determines whether two object instances are equal. This method is overloaded.
- GetBaseException - When overridden in a derived class, returns the Exception that is the root cause of one or more subsequent exceptions.
- GetHashCode - Serves as a hash function for a particular type.
- GetObjectData - When overridden in a derived class, sets the SerializationInfo with information about the exception.
- ToString - Creates and returns a string representation of the current exception.

Boxing and UnBoxing

The conversion of a value type to a reference type is called boxing. The conversion of a reference type to a value type is called unboxing. Objects based on nullable types are boxed only if the object is not null. Therefore, if the nullable type is null (and the *HasValue* property returns *false)*, the object reference returns null and is not boxed. If the nullable type is not null (and the *HasValue* property returns *true*), the underlying type on which the object is based is boxed. However, boxing a non-null nullable value type boxes the value type itself, not the nullable data type that wraps the value type.

TypeForwardedToAttribute Class

The *TypeForwardedToAttribute* This class, which is new to the .NET Framework 2.0, specifies the destination Type in another assembly (and cannot be inherited). You can use this class to move a type from one assembly to another while not disrupting the callers compiled against the original assembly.

2.3 Managing associated data using collections

System.Collections Namespace

The *System.Collections* namespace supports collections, including classes such as *ArrayList*, *BitArray*, *Dictionaries*, and *Stack*. These collections simplify using complex data.

Some classes exposed by the *System.Collections* namespace are :

Classes

Name	Description
ArrayList	Implements the *iList* interface whose size is dynamically increased as required as an index-based collection of objects
BitArray	Manages a compact array of bit values (Booleans). **true** indicates the bit is on (1); **false** indicates the bit is off (0)
CaseInsensitiveComparer	Compares two objects, ignoring the string case
CaseInsensitiveHashCodeProvider	Provides the hash code for an object suing a hashing algorithm, ignoring the string case **Note:** This class is now obsolete and not supported in the .NET Framework 2.0.
CollectionBase	Provides the abstract base class for a strongly typed collection
Comparer	Compares two objects for equivalence. String comparisons are case sensitive
DictionaryBase	Provides the abstract base class for a strongly typed collection of key/value pairs

Hashtable	Represents a collection of key/value pairs organized based on the hash code of the key (items can be retrieved by name or index)
Queue	Represents a first-in, first-out (FIFO) collection of objects
ReadOnlyCollectionBase	Provides the abstract base class for a strongly typed non-generic read-only collection
SortedList	Represents a collection of key/value pairs sorted by the keys and accessible by key and index
Stack	Represents a last-in-first-out (LIFO) non-generic collection of objects

Interfaces

Name	Description
ICollection	Defines size, enumerators, and synchronization methods for all non-generic collections
IComparer	Provides a method that compares two objects
IDictionary	Represents a non-generic collection of key/value pairs
IDictionaryEnumerator	Enumerates the elements of a non-generic dictionary
IEnumerable	Exposes the enumerator, which supports a simple iteration over a non-generic collection
IEnumerator	Provides a simple iteration over a non-generic collection
IEqualityComparer	Provides methods to support the comparison of objects for equality This is new in the .NET Framework version 2.0
IHashCodeProvider	Provides a hash code for an object using a custom hash function **Note:** This interface is now obsolete and not supported in the .NET Framework 2.0
IList	A non-generic collection of objects which can be individually accessed by index

Structures

Name	Description
DictionaryObject	Defines a dictionary key/value pair that can be set or retrieved

ArrayList class
This class implements the *iList* interface to contain unordered objects of any type whose size is dynamically increased as required. If the elements are not sorted, you may need to call the *sort* method before calling operations that require the *ArrayList* to be sorted, such as the *BinarySearch*. Items can be added to the *ArrayList* using the *Add* method to add a single object to the collection or *AddRange* to add several objects (from another array or collection object that supports the *iCollection* interface). The capacity property is automatically set to the number of elements in the *ArrayList* and automatically reallocated when new elements are added to the *ArrayList*. However, it can be explicitly set by setting the *Capacity* property or decreased by calling the *TrimToSize* method. Indexes in the *ArrayList* are zero-based and can be accessed using an integer index. The *ArrayList* accepts null references and can contain duplicate elements.

Collection interfaces

ICollection interface and IList interface

The *ICollection* is the base interface for classes in the *System.Collections* namespace. This interface defines size, enumerators and synchronization methods for all non-generic collections. The *IList* interface is a specialized interface that extends the *ICollection* interface and defines a non-generic collection of objects (values and its members) which can be individually accessed by index (similar to the *ArrayList* class previously described).

Note: If neither the *ICollection* interface nor *IList* interface meet your required collection's requirements, you should use the *ICollection* interface to derive your new class since it offers greater flexibility.

IComparer interface and IEqualityComparer interface

The *IComparer* interface provides a method that compares two objects. Using it with the *Array.Sort* and *Array.BinarySearch* methods, you can customize the sort of a collection of objects. The *IEqualityComparer* interface (new in the .NET Framework 2.0) provides methods to support comparing objects for equality. With the *IEqualityComparer* interface, you can create a custom definition of equality. The collection types that accept this interface are: *Hashtable*, *NameValueCollection* and *OrderedDictionary*.

Note: The *IComparer* interface supports only equality operations. To customize comparisons for sorting or ordering, use the *IComparer* interface.

IDictionary interface and IDictionaryEnumerator interface

The *IDictionary* interface represents the base interface for non-generic collections of key/value pairs. Each element is a key/value pair stored in a *DictionaryEntry* object. Each pair needs a unique key (unless the value is a null reference, then it need not be unique). The keys in the *IDictionary* interface can be enumerated but are not sorted. *IDictionary* interfaces are one of three types: read-only, fixed-size and variable size. The *IDictionaryEnumerator* interface enumerates the elements of a non-generic dictionary.

IEnumerable interface and IEnumerator interface

The *IEnumerable* interface the enumerator, which supports a simple iteration over a non-generic collection. The *IEnumerator* interface provides a simple iteration over a non-generic collection.

Iterators

Iterators are a generalization of pointers that allow programmers to work with data structures in a uniform manner. They are intermediaries between containers and generic algorithms. The five types of iterators are:

- Output: forward moving, may store but not retrieve values, provided by ostream and inserter
- Input: forward moving, may retrieve but not store values, provided by istream
- Forward: forward moving, may store and retrieve values
- Bidirectional: forward and backward moving, may store and retrieve values, provided by list, set, multiset, map and multimap
- Random access: elements accessed in any order, may store and retrieve values, provided by vector, string and array

Hashtable class
The *System.Collections.Hashtable* class represents a collection of key/value pairs organized based on the key's hash code.

CollectionBase class and ReadOnlyCollectionBase class
The *System.Collections.CollectionBase* class provides the abstract base class for a strongly typed collection. The *System.Collections.ReadOnlyCollectionBase* class provides the abstract base class for a strongly typed non-generic read-only collection.

DictionaryBase class and DictionaryEntry class
The *System.Collections.DictionaryBase* class provides the abstract base class for a strongly typed collection of key/value pairs. The *System.Collections.DictionaryEntry* provides a non-generic collection of key/value pairs.

Comparer class
The *System.Collections.Comparer* class compares two objects for equivalence. String comparisons are case sensitive.

Queue class
The *System.Collections.Queue* class represents a first-in, first-out (FIFO) collection of sequential objects.

SortedList class
The *System.Collections.SortedList* class represents a collection of key/value pairs sorted by keys and accessible by key and by index.

BitArray class
The *System.Collections.BitArray* class manages a compact array of bit values (Booleans). **True** indicates the bit is on (1), and **false** indicates the bit is off (0).

Stack class
The *System.Collections.Stack* class represents a last-in-first-out (LIFO) non-generic collection of objects.

2.4 Improving type safety and application performance using generic collections

System.Collections.Generic Namespace

The *System.Collections.Generic* namespace (new to the .NET Framework 2.0) contains interfaces and classes that define generic collections, allowing developers to create strongly typed collections. This namespace provides better safety and performance than non-generic strongly typed collections.
Members exposed by the *System.Collections.Generic* namespace are:

Classes

Name	Description
Comparer Generic	Provides a base class for implementations of the *IComparer* generic interface
Dictionary Generic	Represents a collection of keys and values
Dictionary.KeyCollection Generic	Represents the collection of keys in a Dictionary; cannot be inherited
Dictionary.ValueCollection Generic	Represents the collection of values in a Dictionary; cannot be inherited
EqualityComparer Generic	Provides a base class for implementations of the *IEqualityComparer* generic interface
KeyNotFoundException	Exception thrown when the key specified for accessing an element in a collection does not match any key in the collection
LinkedList Generic	Represents a doubly linked list
LinkedListNode Generic	Represents a node in a *LinkedList;* cannot be inherited

List Generic	Represents a strongly typed list of objects that can be accessed by index; provides methods to search, sort, and manipulate lists
Queue Generic	Represents a first-in, first-out (FIFO) collection of objects
SortedDictionary Generic	Represents a collection of key/value pairs sorted on the key
SortedDictionary.KeyCollection Generic	Represents the collection of keys in a *SortedDictionary;* cannot be inherited
SortedDictionary.ValueCollection Generic	Represents the collection of values in a *SortedDictionary; cannot be inherited*
SortedList Generic	Represents a collection of key/value pairs sorted by key based on the associated *IComparer* implementation
Stack Generic	Represents a variable size last-in-first-out (LIFO) collection of instances of the same arbitrary type

Interfaces

Name	Description
ICollection Generic	Defines methods to manipulate generic collections
IComparer Generic	Defines a method that a type implements to compare two objects
IDictionary Generic	Represents a generic collection of key/value pairs
IEnumerable Generic	Exposes the enumerator, which supports a simple iteration over a collection of a specified type
IEnumerator Generic	Supports a simple iteration over a generic collection
IEqualityComparer Generic	Defines methods to support the comparison of objects for equality
IList Generic	Represents a collection of objects that can be individually accessed by index

Structures

Name	Description
Dictionary.Enumerator Generic	Enumerates the elements of a Dictionary
Dictionary.KeyCollection. Enumerator Generic	Enumerates the elements of a *Dictionary.KeyCollection*
Dictionary.ValueCollection. Enumerator Generic	Enumerates the elements of a *Dictionary.ValueCollection*
KeyValuePair Generic	Defines a key/value pair that can be set or retrieved
LinkedList.Enumerator Generic	Enumerates the elements of a *LinkedList*
List.Enumerator Generic	Enumerates the elements of a List
Queue.Enumerator Generic	Enumerates the elements of a Queue
SortedDictionary.Enumerator Generic	Enumerates the elements of a *SortedDictionary*
SortedDictionary.KeyCollection. Enumerator Generic	Enumerates the elements of a SortedDictionary. KeyCollection.
SortedDictionary.ValueCollection. Enumerator	Enumerates the elements of a *SortedDictionary. ValueCollection*
Stack.Enumerator Generic	Enumerates the elements of a Stack

Collection.Generic interfaces

Generic *IComparable* interface (Refer System Namespace)

The Generic *IComparable* interface defines a generalized comparison method that a value type or class implements to create a type-specific comparison method for ordering instances.

Generic ICollection interface and Generic IList interface
The *ICollection* defines methods to manipulate generic collections; it is the base interface for classes in the *System.Collections.Generic* namespace. The *IList* interface, a more specialized interface, extends the *ICollection*. The *IList* implementation is a collection of values whose members can be accessed by index.

Generic IComparer interface and Generic IEqualityComparer interface
The generic *IComparer* interface defines a method that a type implements to compare two objects. It is used with the *System.Collections.Generic.List.Sort* and *System.Collections.Generic.List.BinarySearch* methods to customize a collection's sort order. It is implemented by the *SortedDictionary* and *SortedList* generic classes. The default implementation of this interface is the *Comparer* class.

The *IComparer* interface supports comparing ordered lists. When using the *Compare* method, if the objects sort the same, this method will return 0. Use the IEqualityComparer generic interface for exact equality comparisons.

Generic IDictionary interface
The *IDictionary* interface represents a generic collection of key/value pairs. With this interface, each element is a key/value pair stored in a *KeyValuePair* object and requires a unique key. The value need not be unique and the value can be a null reference. While the keys and values can be enumerated, the order of the list is not sorted.

Generic IEnumerable interface and Generic IEnumerator interface.
The generic *IEnumerable* interface exposes the enumerator that supports simple iterations over a collection of specified types. The generic *IEnumerator* interface exposes the base interface for all generic enumerators (which Supports a simple iteration over a generic collection). You should use the *ForEach* statement to enumerate the collection rather than directly manipulating the enumeration. Enumerators can read data in the collection but they cannot modify the underlying collection. Because the enumerator is positioned before the first element in the collection, the *Current* property is undefined. You must call the *MoveNext* method to advance the enumerator to the first element of the collection before reading the value of *Current*. When calling the *MoveNext* method and you have passed the end of the collection, the *MoveNext* method returns false and the *Current* property is undefined. You must create a new enumerator instance to access an element in the collection again. The enumerator remains valid if the collection has not been changed. Once the collection is modified (elements are added, modified or deleted), the enumerator becomes invalid.

IHashCodeProvider interface
The *IHashCodeProvider* interface, which was supported in .NET Framework 1.0 and 1.1, is obsolete in 2.0. When programming, you will receive an error during compilation. This interface supplied a hash code for an object, using a custom hash function.

Generic Dictionary

Generic Dictionary class and Generic *Dictionary.Enumerator* structure
The generic *Dictionary* class provides a mapping from a set of keys to a set of values, implemented as a hash table. Each addition to the dictionary consists of a value and its associated key. Each item in the dictionary object is treated as a KeyValuePair structure, representing a value and its key. When enumerating the object, using the *foreach* method, the type of each element in the collection must be supplied.

Generic Dictionary.KeyCollection class and Dictionary.KeyCollection.Enumerator structure
The generic *Dictionary.KeyCollection* class represents a collection of keys in the Dictionary object. The *Dictionary.KeyCollection.Enumerator* enumerates the elements of the the key collection. You should use the *ForEach* statement to enumerate the collection rather than directly manipulating it. Enumerators can read data in the collection but cannot modify the underlying collection. Because the enumerator is positioned before the first element in the collection, the *Current* property is undefined. You must call the *MoveNext* method to advance the enumerator to the first element of the collection before reading the value of *Current*. When calling the *MoveNext* method and you have passed the end of the collection, the *MoveNext* method returns false and the *Current* property is undefined. You must create a new enumerator instance to access an element in the collection again. The enumerator remains valid if the collection has not been changed. If the collection is modified (elements are added, modified or deleted), the enumerator becomes invalid.

Generic Dictionary.ValueCollection class and Dictionary.ValueCollection.Enumerator structure
The generic *Dictionary.ValueCollection* class represents the collection of values in a Dictionary object. The *Dictionary.ValueCollection* refers to the values in the original Dictionary (not a static copy of the object). Therefore, changes to the Dictionary object are immediately reflected in the *Dictionary.ValueCollection* object. The *Dictionary.ValueCollection.Enumerator* enumerates the elements of a *Dictionary.ValueCollection* object. You should use the *ForEach* statement to enumerate the collection rather than directly manipulating it. Enumerators can read data in the collection but cannot modify the underlying collection. Because the enumerator is positioned before the first element in the collection, the *Current* property is undefined.

You must call the *MoveNext* method to advance the enumerator to the first element of the collection before reading the value of *Current*. When calling the *MoveNext* method and you have passed the end of the collection, the *MoveNext* method returns false and the *Current* property is undefined. You must create a new enumerator instance to access an element in the collection again. The enumerator remains valid if the collection has not been changed. If the collection is modified (elements are added, modified or deleted), the enumerator becomes invalid.

Generic Comparer class and Generic EqualityComparer class
The generic Comparer class provides the default implementations of the *IComparer* generic interface. The interface allows you to implement the *Compare* method to compare two objects and return an integer representing the result of the comparison.

The generic *EqualityComparer* class provides a base class for implementing the *IEqualityComparer* generic. The interface allows you to compare equality of objects.

Generic KeyValuePair structure
The Generic KeyValuePair structure is new to the .NET Framework 2.0. This structure defines a key/value pair that can be set or retrieved.

Generic List class, Generic List.Enumerator structure, and Generic SortedList class
The Generic *List* class provides a strongly typed list of objects that can be accessed by index, searched, sorted, and can manipulate lists. The generic *List* structure is new to the .NET Framework 2.0. The *List. Enumerator* structure enumerates the elements of the List. The *List.Enumerator* structure is new to the .NET Framework 2.0. The Generic *SortedList* class provides a collection of key/value pairs sorted by key based on the associated *IComparer* implementation. The Generic *SortedList* class is new to the .NET Framework 2.0.

Generic Queue class and Generic Queue.Enumerator structure
The generic *Queue* class represents a first-in, first-out (FIFO) collection of objects. This class stores elements in a circular array: Objects in the queue are inserted at one end and removed from the other. The object's capacity is automatically increased as required through reallocation.

Generic SortedDictionary class
The generic *SortedDictionary* class provides a collection of key/value pairs sorted on the key. Each element (stored as a key/value pair) can be retrieved as a *KeyValuePair* structure or a *DictionaryEntry* (using the nongeneric *IDictionary* interface).

Generic LinkedList class

Generic LinkedList class
The generic *LinkedList* class represents a doubly linked list. Doubly linked lists indicate that each node in the list points forward to the next element and back to the previous one.

Generic LinkedList.Enumerator structure
The generic *LinkedList* class supports enumerations via the generic *LinkedList.Enumerator* structure and implements the *iCollection* interface.

Generic LinkedListNode class
Each element in the *ListList* is a *LinkListNode*. Insertions into the *LinkedList* object and removals are 0/1 operations.

Generic Stack class and Generic Stack.Enumerator structure
The Generic *Stack* class provides a variable size last-in-first-out (LIFO) collection of instances (implemented as an array) of the same arbitrary type. As elements are added to the Stack class object, the capacity is automatically increased by size reallocation on the internal array.

The Generic *Stack.Enumerator* structure allows you to enumerate the elements of the Stack object. The Stack class accepts null references and allows duplicate elements. You should use the *ForEach* statement to enumerate the collection rather than directly manipulating the enumeration. Enumerators can read data in the collection but cannot modify the underlying collection. Because the enumerator is positioned before the first element in the collection, the *Current* property is undefined. You must call the *MoveNext* method to advance the enumerator to the first element of the collection before reading the value of *Current.* When calling the *MoveNext* method and you have passed the end of the collection, the *MoveNext* method returns false and the *Current* property is undefined. You must create a new enumerator instance to access an element in the collection again. The enumerator remains valid if the collection has not been changed. If the collection is modified (elements are added, modified or deleted), the enumerator becomes invalid.

2.5 Managing data by using specialized collections

System.Collections.Specialized Namespace

The *System.Collections.Specialized* namespace contains special types of classes derived from the *System.Collections* namespace that provide features not available in the base collection classes.

Specialized String classes

StringCollection class
The *StringCollection* class is a type of *System.Collections.Specialized* class that provides a strongly typed string collection. The *StringCollection* class behaves like an *ArrayList* except that, unlike the *ArrayList* which can store items of any type, the *StringCollection* class can only contain items of type String. The same methods are exposed for the *StringCollection* class such as *Add*, *IndexOf* and *Remove.* Since it is type-safe, when storing a collection of strings, you should use the *StringCollection* class, not the *ArrayList* class.

StringDictionary class
The *StringDictionary* class is a type of *System.Collections.Specialized* class that provides a strongly typed, string-based version of the *HashTable* (or an associative array). The *StringDictionary* class behaves like the *HashTable* except that, unlike the *HashTable* which stores values of type Object (you can add any type to a *Hashtable)*, with the *StringDictionary* class, both the keys and values must be of type String.

StringEnumerator class
The *StringEnumerator* class allows you to perform a simple iteration over a *StringCollection*. You should use the *ForEach* statement to enumerate the collection rather than directly manipulating the enumeration. Enumerators can read data in the collection but cannot modify the underlying collection. Because the enumerator is positioned before the first element in the collection, the *Current* property is undefined. You must call the *MoveNext* method to advance the enumerator to the first element of the collection before reading the value of *Current*. When calling the *MoveNext* method and you have passed the end of the collection, the *MoveNext* method returns false and the *Current* property is undefined. Therefore, you must call the *Reset* method followed by the *MoveNext* method to set the position of the collection to the first element. The enumerator remains valid if the collection has not been changed. If the collection is modified (elements are added, modified or deleted), the enumerator becomes invalid.

Specialized Dictionary

HybridDictionary class
The *HybridDictionary* class provides a hybrid implementation of the Dictionary class. It implements the *IDictionary* class using the *ListDictionary* namespace while the list contains 10 or fewer items, and switches to the *HashTable* when the collection contains 10 elements or more. This class uses the optimal data structure for the object's size and switches transparently when the object size grows. It is useful when the number of elements is unknown.

IOrderedDictionary interface and OrderedDictionary class
The *IOrderedDictionary* interface represents an indexed collection of key/value pairs. Each element stored in the in the *IOrderedDictionary* interface is a key/value pair stored in the *DictionaryEntry* structure and can be referenced by its key or index. Each element needs a unique key that is not a null reference. The value need not be unique; it can be a null reference.

The *OrderedDictionary* class represents an indexed collection of key/value pairs that can be referenced by their key or index. Like the *IOrderedDictionary* interface, elements in the *OrderedDictionary* class are key/value pairs stored in the *DictionaryEntry* structure. Elements are not sorted.

ListDictionary class
The *ListDictionary* class implements the *IDictionary* class using a singly linked list. Based on the same logic as in the *HybridDictionary* class, this class is recommended for collections that will contain 10 or fewer elements since it will perform faster than a *HashTable*. The elements are not sorted, the key cannot be a null reference, but the value item can be a null reference.

Named collections

NameObjectCollectionBase class
The *NameObjectCollectionBase* class provides the abstract base class for a collection of String Keys and Object values. The elements can be accessed by the key or the index. The underlying structure of this class is a *HashTable* (each element is a key/value pair). This class' capacity is automatically increased as elements are added.

NameObjectCollectionBase.KeysCollection class
The *NameObjectCollectionBase.KeysCollection* class represents a collection of String keys in the collection.

NameValueCollection class
The *NameValueCollection* class provides a collection of associated String keys and string values (key/value pair elements). Elements can be accessed by the using key or the index. The class is based on the *NameObjectCollectionBase* class. However, unlike the *NameObjectCollectionBase*, this class stores multiple string values under a single key. Therefore, multiple values per key and values can be retrieved by the index or key. This class is used in Web applications to store Headers, Query Strings, and Form data. The capacity of this class is automatically increased as elements are added.

CollectionsUtil class
The CollectionsUtil class allows you to create collections that ignore the string's case.

BitVector32 structure and BitVector32.Section structure
The *BitVector32* structure provides the structure to store Boolean values or small integers in 32 bits of memory. This structure allows you to manage individual bits in larger numbers, thus changing the value of an integer within a number. It can contain either sections for small integers or bit flags for Booleans, but not both. Note that some members are usable for *BitVector32* classes set up as sections, and other members are usable for *BitVector32* classes set up as bit flags. The *BitVector32*.Section structure class represents a section of the *BitVector32* that can contain an integer number. It is created using the *CreateSection* method of the *BitVector32* structure.

2.6 Implementing .NET Framework interfaces to cause components to comply with standard contracts.

System Namespace

The *System* Namespace defines several interfaces. An interface is a set of function definitions you can implement. An interface is defined to enforce a common design pattern among classes that not hierarchically related. An example is the *iDisposable* interface. The *iDisposable* interface contains the *Dispose* method, with which you are probably familiar. The *Dispose* method provides a mechanism for programmers to force an object to perform its cleanup code immediately, rather than waiting for the garbage collector to dispose of the object and execute the cleanup code. Many objects might use this behavior, but many will not need it. Therefore, it has not automatically been added to the *System.Object* base class. However, if you have a class for which you want to use the *Dispose* method, that class can implement the *iDisposable* interface. The class can preserve its inheritance relationship(s) with its base classes while implementing an additional interface.

The .NET architects determined that the following interfaces were common enough to occupy the System namespace:

- IAsyncResult
- ICloneable
- IComparable
- IDisposable
- IFormatProvider
- IFormattable

IComparable interface
Defines a generalized comparison method that a value type or class implements to create a type-specific comparison method.

IDisposable interface
Defines a method to release allocated, unmanaged resources

IConvertible interface
Provides methods to convert the value of an instance of an implementing type to a common language runtime type with an equivalent value. The types are Boolean, SByte, Byte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal, DateTime, Char and String. If the conversion cannot be executed, an InvalidCastException error is thrown.

ICloneable interface
Supports the creation of a new instance of a class with the same value as the original.

IEquatable interface
Provides a generalized method that a value type or class implements to create a type-specific method to determine equality of instances. This interface is new to the .NET Framework 2.0.

IFormattable interface
Allows you to format the value of an object into a string representation.

2.7 Controlling interactions between application components by using events and delegates

Delegate class
A *Delegate* represents a pointer to an individual object method or a static class method. Within the .NET Framework, callbacks from one object to another are supported by the *Delegate* class. Unlike other classes, a *Delegate* class has a signature, like a function. Programmers do not typically use the *Delegate* class directly, instead using a wrapper provided by the programming language. Thus, for an object to receive an event, it provides the sender a delegate. The sender calls the function on the delegate to signal the event.

EventArgs class
The *EventArgs* is the base class for classes containing event data. This class is used by events that do not pass state information to an event handler when an event is raised.

EventHandler delegates
If you have been developing applications in ASP.NET or Windows Forms, you are familiar with the generic delegate type *EventHandler*. The *EventHandler* is contained by the *System* Namespace and has the following signature:

VB
Public Delegate Sub EventHandlerName(ByVal sender as Object, ByVal e As EventArgs)

or

C#
public delegate void eventhandlername(object sender, eventargs e)

The first parameter, *sender*, is based on the generic object type and passes a reference to the event source object. The *e* parameter is an object of *EventArgs*, or the actual event data (if available) that derives its class from *EventsArgs* type. If there is no additional parameter information, the value of the *e* object is set to *EventArgs.Empty* or *Nothing*. When parameters are passed in the *e* object, the object must be created from a class that derives from the *EventArgs* class.

Implementing service processes, threading and application domains in a .NET Framework application

3.1 Use Implement, install, and control a service

System.ServiceProcess namespace

Windows services are processes that run in the background, in their own session without a user interface. The.NET Framework makes it easy to develop applications to run as Windows services. The *System.Service* Process namespace provides the classes to implement, install and control Windows services. Windows services are defined as "long-running executables that run without a user interface."

Note: Services are installed using an installation utility such as **InstallUtil.exe.** The *System.ServiceProcess* namespace contains installation classes to write required system information to the registry. For example: **InstallUtil.exe [ServiceExeFile]**

Some key features about services are:

- The executable file created by the service application project must be installed and started before the project can be debugged by attaching to the service's process.
- The service must be installed and registered using installation components, thereby creating an entry in the Windows Service Control Manager.
- The *Main* method of the service application must contain the *Run* command for the service.
- Service applications run in their own window station that cannot interact with the client user interface (e.g. display dialog boxes, error messages). Therefore, errors or event messages must be written to the Windows event log (or developer-created log file).
- Service applications are executed in their own security model using a specified user account, not the current user account.

Members exposed by the *ServiceProcess* class are :

Classes

Name	Description
ServiceBase	Provides the base class for a service that will exist as part of a service application.
ServiceController	Represents the Windows service and allows you to connect to a running or stopped service, get information about it and control it.
ServiceControllerPermission	Allows control of code access security permissions for service controllers.
ServiceControllerPermissionAttribute	Allows declarative service controller permission checks.
ServiceControllerPermissionEntry	The smallest unit of a code access security permission that is set for the *ServiceController.*
ServiceControllerPermission EntryCollection	A strongly-typed collection of *ServiceControllerPermissionEntry* objects.
ServiceInstaller	Installs a class that extends *ServiceBase* to implement a service (class is called by the install utility when installing a service application.)
ServiceProcessDescriptionAttribute	Specifies a description for a property or event.
ServiceProcessInstaller	Installs an executable containing classes that extend *ServiceBase* (is called with the service application is installed).
TimeoutException	The exception that is thrown when a specified timeout has expired.

Structures

Name	Description
SessionChangeDescription	Provides the reason for a Terminal Services session change. This is new in the .NET Framework version 2.0.

Enumerations

Name	Description
PowerBroadcastStatus	The system's power status
ServiceAccount	A service's security context (which defines its logon type)
ServiceControllerPermissionAccess	Access levels used by *ServiceController* permission classes
ServiceControllerStatus	Indicates the current state of the service
ServiceStartMode	Indicates the start mode of the service
ServiceType	The type of the service
SessionChangeReason	The reason for a Terminal Services session change notice. This is new in the .NET Framework version 2.0.

Inherit from ServiceBase class
The *ServiceBase* class is the base class for Windows services. To create a service, your class will inherit (directly or indirectly) from this class. You will define specific actions for the *OnStart*, *OnPause*, *OnStop* and *Continue* methods, as well as custom actions when the system shuts down.

ServiceController class and ServiceControllerPermission class
The *ServiceController* class allows you to connect to an existing service. You can read information about the service and manipulate it (start, stop, pause, continue or execute custom commands on the service). When you create an instance of the *ServiceController* object, you must set the computer name and service name to which you want to connect. The *MachineType* property is set to the local computer by default and must be modified only if you are connecting to another computer. Typically, the *ServiceController* component is used in an administrative capacity.

Note: Because the Service Control Manager (SCM) does not support custom commands, you must use the *ServiceController* class to send custom commands to the Web service.

The *ServiceControllerPermission* class provides programmatic control of code access security permissions for the *ServiceController* (adding permission access, removing permission access, etc).

ServiceInstaller and ServiceProcessInstaller class
The *ServiceInstaller* is called by the service installation utility when the service is being installed to write registry values associated with the service to the subkey in the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services registry key. The subkey identifies the name of the service and the executable or .dll file to which the service belongs.

The *ServiceProcessInstaller* class is called by the service installation utility when the service is being installed to install the executable containing the classes that extend the *ServiceBase*. It writes registry values for the service being installed. One property you can specify when installing a service is what account the service application runs under (if not the current logged-in account). You can specify the username and password of

an account, and specify that the service run under the computer's System account, a local or network service account. Remember that the computer system account is different from the Administrator account.

SessionChangeDescription structure and SessionChangeReason enumeration
The *SessionChangeDescription* structure (new in the .NET Framework version 2.0) identifies the reason for a Terminal Services session change while the *SessionChangeReason* enumeration (new in the .NET Framework version 2.0) provides the reason for a Terminal Services session change notice (for example, the console session has been connected or disconnected, the session has been locked or unlocked, the user has logged on or logged off, etc.).

3.2 Develop multithreaded .NET Framework applications

System.Threading namespace

The *System.Threading* namespace provides interfaces and classes that enable multithreaded applications. Multithreading techniques allow developers to create scalable, efficient applications that can execute multiple tasks concurrently.

Thread class
The Thread class is the core class providing methods to create threads, and control threads (for example: suspend, stop and destroy), set their priority and get the threads' status. Using this class, a process can create one or more threads that execute part of the program code associated with the process. When the thread has been started, it will continue to execute until it has finished.

The *Thread* constructor initializes a new instance of the *Thread* class. This constructor is overloaded; therefore, the following signatures are supported:

- Thread (ParameterizedThreadStart) - Indicates a delegate object to be passed to the thread when the thread is started
- Thread (ThreadStart)
- Thread (ParameterizedThreadStart, Int32) - Indicates a delegate object to be passed to the thread when the thread starts; specifies a maximum stack size
- Thread (ThreadStart, Int32) - Specifies a maximum stack size for the thread

Some of the public properties exposed by the *Thread* class are:

- ApartmentState - Assigns or returns the apartment state of the thread.
- CurrentContext - Returns the current context in which the thread is executing. This property is static.
- CurrentCulture - Assigns or returns the culture for the current thread.
- CurrentPrincipal - Assigns or returns the thread's current principal used for role-based security. This property is static.
- CurrentThread - Returns the currently executing thread. This property is static.
- CurrentUICulture - Assigns or returns the current culture used by the Resource Manager to look up culture-specific resources at run time.

- ExecutionContext - Returns an *ExecutionContext* object containing information regarding the various contexts of the thread.
- IsAlive - Returns a value indicating the execution status of the thread (is it currently executing).
- IsBackground - Assigns or returns a value indicating whether or not a thread is a background thread.
- IsThreadPoolThread - Returns a value indicating whether or not a thread belongs to the managed thread pool.
- ManagedThreadId - Returns a unique identifier for the managed thread.
- Name - Assigns or returns the name of the thread.
- Priority - Assigns or returns a value indicating a thread's scheduling priority.
- ThreadState - Returns a value containing the states of the thread.

Some of the public Methods exposed by the *Thread* class are:

- Abort - Raised a *ThreadAbortException* in the thread indicating that the thread should be aborted (thus terminating the thread).
- AllocateDataSlot - Allocates an unnamed data slot on all the threads. This method is static.
- AllocateNamedDataSlot - Allocates a named data slot on all threads. This method is static.
- BeginCriticalRegion - Signals host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception might jeopardize other tasks in the application domain. This method is static.
- BeginThreadAffinity - Signals host that managed code is about to execute instructions that depend on the identity of the current physical operating system thread. This method is static.
- EndCriticalRegion - Signals a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exceptions are limited to the current task. This method is static.
- EndThreadAffinity - Signals a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception are limited to the current task. This method is static.
- FreeNamedDataSlot - Eliminates the association between a name and a slot for all of the threads in the current process. This method is static.
- GetApartmentState - Gets an *ApartmentState* value indicating the apartment state.
- GetCompressedStack - Gets a *CompressedStack* object that can capture the stack for the thread.
- GetData - Returns the value from the specified slot on the current thread, within the current thread's current domain. This method is static.
- GetDomain - Gets the current domain in which the thread is running. This method is static.
- GetDomainID - Gets a unique application domain identifier. This method is static.
- GetNamedDataSlot - Gets a named data slot. This method is static.

- Interrupt - Raises the *ThreadInterruptedException* when the thread is in a blocked state (*WaitSleepJoin* thread state), thus interrupting the thread.
- Join - Blocks the calling thread until a thread terminates.
- MemoryBarrier - Synchronizes memory access as follows: The processor executing the current thread cannot reorder instructions so memory accesses before the call to MemoryBarrier execute after memory accesses that follow the call to MemoryBarrier. This method is static.
- ResetAbort - Cancels an Abort requested for the thread. This method is static.
- Resume - Resumes a thread that was previously suspended.
- SetApartmentState - Sets the apartment state of a thread before it is started.
- SetCompressedStack - Applies a captured *CompressedStack* to the current thread.
- SetData - Sets the data in the specified slot on the currently running thread. This method is static.
- Sleep - Blocks the current thread for a specified number of milliseconds. This method is static.
- SpinWait - Causes a thread to wait the number of times indicated by the iterations parameter. This method is static.
- Start - Starts the thread to be scheduled for execution.
- Suspend - Suspends the thread (unless it is already suspended, in which case it has no effect).
- ToString - Gets a String that represents the current Object.
- TrySetApartmentState - Sets the apartment state of a thread before it is started.
- VolatileRead - Gets the value of a field. The value is the latest written by any processor in a computer, regardless of the number of processors or the state of processor cache. This method is static.
- VolatileWrite - Writes a value to a field immediately, so the value is visible to all processors in the computer. This method is static.

ThreadPool class
The *ThreadPool* class provides a pool of threads used to post work items, process asynchronous Input/Output operations, wait on behalf of other running threads and process timers. *Threadpools* are helpful because many threads spend considerable time in a sleeping state (either waiting for an event to occur or intentionally sleeping only to awaken periodically to execute).

ThreadStart delegate and ParameterizedThreadStart delegate
The *ThreadStart* delegate simply represents a method that executes on a thread. The thread will begin executing once the *System.Threading.Thread.Start* method is called. The *ParameterizedThreadStart* delegate (new to the .Framework 2.0) also represents a method that executes on the thread that allows an object to pass that contains data for the thread procedure.

Timeout class, Timer class, TimerCallback delegate, WaitCallback delegate, aitHandle class, and WaitOrTimerCallback delegate
The *Timeout* class contains a constant used by the *Threading* namespace to specify an infinite amount of time. *Infinite* is the only member contained by the *Timeout* class used to specify the waiting period. Its value is a constant which accepts an integer timeout value. The *Timeout* class also has several public methods inherited from the *Object* class (namely, *Equals*, *GetHashCode*, *GetType*, *ReferenceEquals*, and *ToString*).

The *Timer* class provides a mechanism to fire off an asynchronous call to execute a method based on an amount of time. The *TimerCallback* delegate specifies the actual method to execute when the timer fires. The delegate must be specified when the timer is initially created and cannot be modified. The method that is executed is not executed on the thread that created the timer, but on a *ThreadPool* thread handled by the system. Also, when the *Timer* is created, you also specify how long to wait until the *Timer* starts and how long to wait between subsequent executions. These settings can later be modified using the *Change* method (the *Timer* can also be disabled using this method).

The *WaitHandle* is signaled once the timer has been disposed (when manually disposed using the *Dispose* method or by the garbage collector). The *WaitHandle* class manages operating system objects waiting for exclusive access to shared resources. Typically, it is used as a base class for synchronization objects.

Note: When using a *Timer* object, you must keep a reference to the object (even if it is still active) in order to avoid having the garbage collector dispose of the object.

The *TimerCallback* delegate represents the method that handles calls from the *Timer* object. The *TimerCallback* delegate executes the method once after the start time has elapsed, then executes the method once per timer interval until the *Dispose* method is called on the *Timer* or the *Timer.Change* method is called passing the interval value of Infinite. The *WaitCallback* delegate represents a callback method to be executed by a *Threadpool* thread. The *WaitOrTimerCallback* delegate represents a method called when the *WaitHandle* is signaled or when it times out.

ThreadState enumeration and ThreadPriority enumeration
While the thread is executing, its state is available from the *ThreadState* property. All possible execution states are represented. Therefore, once the thread is created, it is in one of the states displayed below until it terminates.

Thread State Members

Aborted	The thread is now dead but its state has not yet changed to Stopped
AbortRequested	The *Thread.Abort* method has been invoked on the thread and it has not yet received the pending *System.Threading.ThreadAbortException* that will attempt to terminate it
Background	The thread is executed as a background thread. This state is controlled by setting the *Thread.IsBackground* property
Running	The thread was started. The thread is not blocked and there is no pending *ThreadAbortException*
Stopped	The thread was stopped
StopRequested	The thread is being requested to stop (for internal use only)
Suspended	The thread has been suspended
SuspendRequested	The thread is being requested to suspend
Unstarted	The *Thread.Start* method has not been invoked on the thread

WaitSleepJoin	The thread is blocked (as a result of calling *Thread.Sleep* or *Thread.Join*, of requesting a lock or of waiting on a thread synchronization object such as *ManualResetEvent*)

A thread can be in more than one state at a time.

The *ThreadPriority* enumeration indicates the scheduling priority of a *Thread* (all possible priority values are represented). Every thread is assigned a priority (by default they are assigned a *Normal* priority if not indicated otherwise). The priority of a *Thread* will not affect the *Thread's* state.

Thread Priority Members

AboveNormal	Can be scheduled after threads with Highest priority but before those with Normal priority.
BelowNormal	Can be scheduled after threads with Normal priority but before those with Lowest priority.
Highest	Can be scheduled before threads with any other priority.
Lowest	Can be scheduled after threads with any other priority.
Normal	Can be scheduled after threads with *AboveNormal* priority but before those with *BelowNormal* priority. This is the default priority.

ReaderWriterLock class
The *ReaderWriterLock* class allows you to lock access to resources for readers and writers separately. That is, you can lock single writers to access the data in a resource (for a single thread) or lock multiple readers to access data in a resource (for multiple threads) at the same time. A single thread can hold a write block or a reader block but not both simultaneously. Therefore, writer threads and reader threads are queued separately. Because threads can be queued (waiting for the previous lock to be released), most methods for the *ReaderWriterLock* class accept time out values (expressed in milliseconds).

AutoResetEvent class and ManualResetEvent class
The *AutoResetEvent* class notifies a waiting thread that an event has occurred. This class allows threads to communicate with one another by signaling. A Thread waits for a signal by calling the *WaitOne* method of *AutoResetEvent* class, When the Thread controlling a resource is ready to make the resource available, it calls the *Set* method of *AutoResetEvent* class.

IAsyncResult interface (Refer System namespace)
The *IAsyncResult* interface represents the status of an asynchronous operation. This interface is implemented by classes whose methods need to operate asynchronously. Implementing this interface allows objects to store state information and allows threads to be signaled when operations on shared resources complete.

EventWaitHandle class, RegisterWaitHandle class, SendOrPostCallback delegate, and IOCompletionCallback delegate

The *EventWaitHandle* class (new to the .NET Framework 2.0) represents a thread synchronization event that allows threads to communicate with one another by signaling. One or more threads will block the *EventWaitHandle* until the blocked thread is release by another thread calling the *Set* method.

The *RegisterWaitHandle* class is handle that has been registered when calling the *RegisterWaitForSingleObject* method (which registers a delegate waiting for a *WaitHandle*).

The *SendOrPostCallback* delegate (new to the .NET Framework 2.0) represents a method to execute when a message is to be dispatched to a synchronization context.

The *IOCompletionCallback* delegate receives an error code, number of bytes and overlapped value type when an I/O operation completes on a thread pool.

Note: The *IOCompletionCallback* delegate is not CLS compliant.

Interlocked class

The *Interlocked* class provides a mechanism for applying atomic operations to variables that are being shared by multiple threads. When the schedule switches contexts, a variable could be updated by one thread while other threads are accessing the same variable, or when multiple threads are executing concurrently on separate processors. While the *Interlocked* class works with only a limited number of .NET types, this method does help protect against errors. This class provides only a limited number of methods. Specifically, it can be used to Increment a variable, Decrement a variable, Exchange values of specified variables, Read a number, and Add two integers (replacing the first with the sum). Because these operations are performed as atomic operations, they are executed as one step rather than three, as on most computers.

ExecutionContext class, HostExecutionContext class, HostExecutionContextManager class, and ContextCallback delegate

The *ExecutionContext* class (new to the .NET Framework 2.0) manages the execution context of the current thread by providing a single container for all the information relevant to a logical thread's execution and by providing the capability for code to capture and transfer context across user-defined asynchronous points (managed by the Common Language Runtime). This includes security context, call context, synchronization context, localization context and transaction context.

Note: An *ExecutionContext* associated with one thread cannot be set on another thread. Attempting to perform this operation will throw an exception. However, you can make copy of the *ExecutionContext* object.

The *HostExecutionContext* class (new to the .NET Framework 2.0) encapsulates/propagates the host execution context across threads.

The *HostExecutionContextManager* class (new to the .NET Framework 2.0) allows the CLR (Common Language Runtime) host to participate in execution context. Specifically, the CLR would call the manager when the *ExecutionContext.Run* method is called (which runs a method in the current execution context on the current thread), thereby allowing the current host to participate in the flow of the execution context.

The *ContextCallback* delegate (new to the .NET Framework 2.0) represents a method to be called within the new context. This method is used by the *ExecutionContext.Run* method and *SecurityContext.Run* method. Once the method has completed, the context is then restored to the state it was in before the method was called.

LockCookie structure, Monitor class, Mutex class, and Semaphore class
The *LockCookie* structure is a value type that provides defines an object lock that implements single writer/multiple reader semantics.

The *Monitor* class allows you to synchronize access to objects by granting an object lock to an object on a single thread. Objects locks can restrict access to blocks of code (called critical sections). While one thread owns the lock for an object, no other threads can acquire that lock. The *Monitor* class locks reference type objects, not value types.

The *Mutex* class is a synchronization primitive used for interprocess synchronization to grant exclusive access to a shared resource to one thread. When a thread acquires a *Mutex* object, other threads are suspended and must wait until the first thread releases the *Mutex*.

The *Semaphore* class (new to the .NET Framework 2.0) is used to throttle usage of a resource by limiting the number of threads that can access the resource or pool of resources concurrently. When a new instance of a *Semaphore* is created, you specify the number of concurrent used slots and number of maximum slots in the kernel object.

3.3 Create a unit of isolation for common language runtime in a .NET Framework application by using application domains

System namespace

Application domains provide isolated environments for applications to execute, providing the ability to call external assemblies while providing optimal security and performance. An application domain is a logical container allowing multiple assemblies to run within a single process. However, assemblies are not permitted to access other assemblies' memories, ensuring they are secure. Multiple assembles can be executed in separate application domains without having to launch separate processes. Another advantage to using application domains rather than launching additional processes is that processes are managed by the operating system but application domains are managed by the CLR.

Create an application domain.

Application domains are created using the *System.AppDomain* class.

An instance of this class is created, then an assembly is executed within that domain. To create an application domain, you call the *AppDomain.CreateDomain* method. This method is overloaded so there are multiple signatures that can be used to create a new application domain. Once you have created the new application domain, you can launch the assembly within that domain by calling the *ExecuteAssembly* method and specifying the complete file path the assembly as displayed below:

VB

```
Dim myDomain As AppDomain = AppDomain.CreateDomain("MyDomain")
myDomain.ExecuteAssembly("Assembly.exe")
```

or

C#

```
AppDomain mydomain = AppDomain.CreateDomain("MyDomain");
mydomain.ExecuteAssembly("Assembly.exe");
```

The ExecuteAssembly method has overloads (multiple signatures) that allow the capability to pass command-line arguments. Another way to execute assemblies is to reference the assembly by name using the *ExecuteAssemblyByName* method of the *AppDomain* class as displayed below:

VB

```
Dim myDomain As AppDomain = AppDomain.CreateDomain("MyDomain")
myDomain.ExecuteAssemblyByName("Assembly")
```

or

C#

```
AppDomain mydomain = AppDomain.CreateDomain("MyDomain");
mydomain.ExecuteAssemblyByName("Assembly");
```

Unload an application domain.

Application domains can be unloaded at any time, thus freeing up resources. To unload an application domain, call the *Unload* method as displayed below:

VB

```
Dim myDomain As AppDomain = AppDomain.CreateDomain("MyDomain")
AppDomain.Unload(myDomain)
```

or

C#

```
AppDomain mydomain = AppDomain.CreateDomain("MyDomain");
AppDomain.Unload(mydomain);
```

In.NET Framework 2.0, a thread is dedicated to unloading application domains. Therefore, once a thread calls the *Unload* method, the domain is marked for unloading and the dedicated thread trying to unload the domain and all threads in the domain are aborted. Three possible exceptions could be thrown when attempting to unload the domain. They are: ArgumentNullException, in which the domain was a null reference; CannotUnloadAppDomainException, in which the domain could not be unloaded; and Exception, in which an error occurred during the unload process.

Configure an application domain.

An application domain can be configured using the *AppDomainSetup* class.
When changing the properties of the *AppDomainSetup* class, the existing *AppDomain* is not affected. Only newly created *AppDomains* are affected once the *CreateDomain* method is called passing the *AppDomainSetup* as a parameter. Properties of the application domain are configured for *AppDomainSetup* object, then passed to the *AppDomain.CreateDomain* method along with the *Evidence* object.

Retrieve setup information from an application domain.

Setup information from the current application domain can be retrieved using the *AppDomain.CurrentDomain.SetupInformation* object. The *CurrentDomain* property returns current application information for the current *Thread*. The *AppDomain.CurrentDomain.SetupInformation* object returns the application domain configuration for this instance of the application domain.

Load assemblies into an application domain

Assemblies are loaded into an application domain using the *AppDomain.Load* method. This method is overloaded so multiple signatures are available to call this method. The assembly can be loaded passing the assembly name, the assembly display name, or the common object file format (COFF)-based image that contains the emitted assembly. Other signatures also provide the ability to pass the *Evidence* object.

Embedding configuration, diagnostic, management, and installation features into a .NET Framework application

4.1 Embed configuration management functionality into a .NET Framework application

System.Configuration Namespace

The *System.Configuration* namespace provides a programming model to handle application configuration information. This namespace is the repository for all the classes, interfaces, delegates and enumerations that developers can use for application configuration. The *System.Configuration* namespace allows you to set and retain application settings without knowing the values in advance. In the .NET Framework 2.0, you can manage the configuration settings in an object-oriented manner. Another major benefit is that you can read, write and modify settings without having to use the Windows Registry, for a more secure application model and greater cross-platform compatibility.

Configuration class and ConfigurationManager class

The *Configuration* class (new to the .NET Framework 2.0) represents the configuration file for the particular physical entity such as a computer, or a logical entity such as an application or resource (located on the local system or a remote system).

Note: If no configuration exists for the specified entity, the *Configuration* object represents the default configuration settings defined by the machine.config file.

The *ConfigurationManager* class provides programmatic access to the configuration files for an application. The *ConfigurationManager* class typically retrieves or stores information for Winform or console applications. While it can be used for ASP.NET applications, Web applications should use the *WebConfigurationManager* class.

You must use the open methods made available by the *WebConfigurationManager* object (for Web applications) or by *ConfigurationManager* object (for client applications) to access the *Configuration* class. These methods return a handle to the *Configuration* object. You can then use the methods and properties of the *Configuration* object to manage the configuration settings. Configuration settings are stored within similarly grouped settings. To read configuration information, use the *GetSection* or the *GetSectionGroup* methods (new to the .NET Framework 2.0) to return specific configuration information by the *ConfigurationSection* object (described below).

ConfigurationElement class, ConfigurationElementCollection class, and ConfigurationElementProperty class

The *ConfigurationElement* class (new to the .NET Framework 2.0) represents a configuration element in a configuration file as an XML element or section. You do not create an instance of the *ConfigurationElement* object as it is an abstract class.

The *ConfigurationElementCollection* class (new to the .NET Framework 2.0) is a configuration element in a configuration file that contains a collection of child elements. This class allows you to add new *ConfigurationElement* elements to a *ConfigurationSection*.

The *ConfigurationElementProperty* class (new to the .NET Framework 2.0) specifies the property of the configuration object. The *Validator* property allows you to validate the *ConfigurationElementProperty* by returning a *ConfigurationValidatorBase* object.

ConfigurationSection class, ConfigurationSectionCollection class, ConfigurationSectionGroup class, and ConfigurationSectionGroup Collection class

The *ConfigurationSection* class (new to the .NET Framework 2.0) represents a section in the configuration file. It allows you to implement a custom section type. This class can be extended to provide custom handling for programmatic access to custom configuration sections. Custom configuration sections can be created using a programmatic coding model or declarative/attributed coding model. Using the programmatic model requires you need to create a property for each section attribute to get/set its value and add it to the underlying *ConfigurationElement* base class. The declarative/attributed model allows you to define a section attribute by using a property and setting its attributes.

The *ConfigurationSectionCollection* class (new to the .NET Framework 2.0) represents a collection of related sections (*ConfigurationSection* objects) in the configuration file. You can iterate through the *ConfigurationSectionCollection* object to get a handle on a *ConfigurationSection* object.

The *ConfigurationSectionGroup* class (new to the .NET Framework 2.0) represents a group of related sections (*ConfigurationSection* objects) in the configuration file. Related sections are typically grouped together.

The *ConfigurationSectionGroupCollection* class (new to the .NET Framework 2.0) represents a collection of *ConfigurationSectionGroup* objects. This class iterates through the collection of *ConfigurationSectionGroup* objects. The collection can be accessed using the *SectionGroups* property.

Implement ISettingsProviderService interface

The *ISettingsProviderService* interface (new to the .NET Framework 2.0) enables an application to define an alternate application settings provider. This interface exposes only one method, *GetSettingsProvider*, which returns the settings provider compatible with the specified settings property.

Implement IApplicationSettingsProvider interface

The *IApplicationSettingsProvider* interface (new to the .NET Framework 2.0) defines the extended capabilities for client-based application settings providers. With this interface, programmers can create a custom settings provider for customized application settings. This interface, derived from the *SettingsProvider* class (providing basic storage and retrieval capabilities), allows programmers to provide additional standardized functionality. This interface contains only three methods which are specifically designed to assist in handling application version handling.

They are:

GetPreviousVersion	Gets the value of the indicated settings property for a previous version of the same application
Reset	Resets the application settings associated with the indicated application to their default values
Upgrade	Indicates to the provider that the application has been upgraded (therefore, the provider can upgrade its stored settings if appropriate)

ConfigurationValidatorBase class

The *ConfigurationValidatorBase* interface (new to the .NET Framework 2.0) is a base class for deriving a validation class to create a custom validator. The custom validator can then be used to verify object values.

4.2 Create a custom Microsoft Windows Installer for the .NET Framework components by using the System. Configuration.Install namespace, and configure the .NET Framework applications by using configuration files, environment variables, and the .NET Framework Configuration tool (Mscorcfg.msc)

Installer class

The *Installer* class is the base class that provides a foundation for custom installations in the .NET Framework. Other specific installers in the .NET Framework are the *AssemblyInstaller* and *ComponentInsaller*. To use the *Installer* class, take the following steps:

1. Inherit the *Installer* class
2. Override the *Install*, *Commit*, *Rollback*, and *Uninstall* methods
3. Add the *RunInstallerAttribute* to your derived class and set it to *true*
4. Put your derived class in the assembly with your application to install
5. Invoke the installers (e.g use InstallUtil.exe)

When using the *Installer* class, the entire installation succeeds or fails (when the *Commit* method is called at the end of the installation). If the install fails, the *Installer* class will undo any changes that have been made. Some methods made available by the *Installer* class are:

- *Commit* – Signals that the installation was a success and changes should be persisted
- *Rollback* – Signals that an error occurred during the installation and that all modifications should be undone
- *Uninstall* – Allows you to completely undo a previously successful installation

AssemblyInstaller class

The *AssemblyInstaller* class is part of the *System.Configuration.Install Namespace* that loads an assembly and runs installers on it. This class can launch an installer programmatically.

ComponentInstaller class

The *ComponentInstaller* class is part of the *System.Configuration.Install Namespace* that specifies an installer that copies properties from a component at the time of installation. This class can launch an installer programmatically. When inheriting from the *ComponentInstaller* class, you must override the *CopyFromComponent* method and may need to override the *Install* and *UnInstall* methods.

ManagedInstallerClass class

The *ManagedInstallerClass* class (derived from the System.Configuration.Install Namespace) is meant to support the .NET Framework infrastructure. Programmers should not use it within their code.

InstallContext class

The *InstallerContext* class (derived from the System.Configuration.Install Namespace) contains information about the current installation. The *InstallerContext* object is created by an installation executable that installs assemblies (e.g., InstallUtil.exe). When the installation executable is run, it invokes the *InstallerContext* constructor passing any parameters or default log-file path information. Before the methods of the *Installer* object are called, the installation program sets the *Context* property of the *Installer* to the newly created instance of the *InstallerContext*. If the *Installer* contains an installer collection (in the *Installers* property), the *Context* property of each contained *Installer* is also set.

InstallerCollection class

The *InstallerCollection* class contains a collection of installers to be run during an application/assembly installation. Installers can be added to the collection with the *Add* method (to add a single installer to the collection), the *AddRange* method (to add multiple installers to the collection), or the *Insert* method (to add a single installer to the collection at a specified index).

InstallEventHandler delegate

The *InstallEventHandler* delegate represents a method that will handle events of the *Installer* class, including: *BeforeInstall*, *AfterInstall*, *Committing*, *Committed*, *BeforeRollback*, *AfterRollback*, *BeforeUninstall*, and *AfterUninstall*. When the *InstallEventHandler* delegate is created, the method that will handle the event is specified. To associate one of these events to your custom event handler, you need to add an instance of the delegate to the event. Your event handles will then be called whenever the event is triggered (unless the delegate is removed).

Configure a .NET Framework application by using the .NET Framework Configuration tool (Mscorcfg.msc)

The .NET Framework 2.0 Configuration tool enables programmers to manage most aspects of the assembly configuration (the assembly cache, assembly configuration, code access security policies, remote services and individual programs). Like the .NET Framework 1.1 Configuration tool, the .NET Framework 2.0 Configuration tool runs as a Microsoft Management Console snap-in. The details of these tasks are outside the scope of this book but should be reviewed before the exam. For more information, refer to the help provided by the .NET Framework 2.0 Configuration tool, the book.*NET Framework 2.0 Application Development Foundation* training kit, or Microsoft's Web support.

4.3 Manage an event log by using the System. Diagnostics namespace

Write to an event log

Programmers know the Windows Event Log, which allows you to record information regarding a software application (or hardware) that might be useful in tracking, supporting and diagnosing an application's state, issues and events. The *EventLog* class (contained within the *System.Diagnostic* namespace) allows programmatic reading, writing or deleting event logs; creating or deleting event sources; or responding to event entries. You can create an event log while creating an event source.

When writing to an event log, you need to specify or create a new event *Source*. The *Source* object will register the application with the event log as a valid entry. The name assigned to the *Source* object must be unique and not identified as another *Source* on the same system. However, a single event log can be associated with multiple sources.

Note: The *Source* is required only when writing to the event log but not when reading it. The *WriteEntry* method provides a mechanism to write to the *EventLog*. This method is simple You need only provide the entry string and entry type but it provides 10 overloads to allow more sophisticated logging when required.

You will be able to write only one event at a time with the *Source* object. When using the *CreateEventSource* method, if the log specified does not exist on the current system, the log will automatically be created and the application will be registered as the *Source*.

You can also specify *EventLogEntryType* to indicate if the entry is an error, warning or information entry type. You can specify the *EventID* displayed in the Type column and Category parameters displayed in the Category column, and attach binary data.

An example of writing to the event log is shown in the "Create a new event log" section.

Read from an event log

To read from a log, you must set the *Log* name and the *MachineName* properties for the *EventLog* object. If you do not specify the *MachineName*, the local computer is assumed (specified as "."). After you have instantiated a handle on an *EventLog* object, you can use the Entries property of the *EventLog* to return an *EventLogEntryCollection* of *EventLogEntry* objects and then iterate through the entries. An example of reading from the event log is displayed below:

VB

```
Dim myLog As New EventLog("LogExample")
For Each myLogEntry As EventLogEntry in myLog.Entries
        Console.WriteLine(myLogEntry.Source & "-" & myLogEntry.Message)
Next
```

or

C#

```
EventLog myLog = new EventLog("LogExample");
Foreach (EventLogEntry myLogEntry in myLog.Entries)
{
        Console.WriteLine(myLogEntry.Source & "-" & myLogEntry.Message);
}
```

You can also retrieve a collection of logs rather than a single log and its entries using the *GetEventLogs* method.

Create a new event log

To create an event log, you simply need to create a new *EventLog* object and specify the *Source* property. An example of this code is displayed below:

VB

```
Dim myLog As New EventLog("LogExample")
myLog.Source = "LogExample"
myLog.WriteEntry("My first log entry", EventLogEntryType.Information)
```

or

C#

```
EventLog myLog = new EventLog("LogExample");
myLog.Source = "LogExample";
myLog.WriteEntry("My first log entry", EventLogEntryType.Information);
```

You can also use the *EventLog* class to create custom events logs viewable from the system event viewer.

Note: Windows 2000 and XP have three default logs: Application, Security, and System. Other applications may also create unique event logs. For example, the Office 2007 beta installation creates two new event logs: "Office Diagnostics" and "Office Sessions" and Internet Explorer 7 (Beta 3) creates an "Internet Explorer" event log.

Event logging uses disk space and processor time. If your log becomes full, your code will throw exceptions when trying to write additional entries to the log. Use the event log judiciously for essential information and error logging. Do not use the *EventLog* object in a partial trust setting; doing so introduces a security vulnerability.

4.4 Manage system processes and monitor the performance of a .NET Framework application by using the diagnostics functionality of the .NET Framework 2.0

Get a list of all running processes

A process is basically an isolated application, or task, using one or more threads. An instance of a Process object can be referring to a process running on a local machine or on a remote machine. One of the following methods can be used to get a list of processes:

- *GetCurrentProcess* – returns a new process component and associates it with the process resource running the calling application
- *GetProcessById* – creates a new process component and associates it with the process resource specified
- *GetProcessesByName* – creates an array of new Process components and associates them with the existing process resources that all share the specified process name
- *GetProcesses* - creates an array of new Process components and associates them with the existing process resources

Retrieve information about the current process

The *GetCurrentProcess* method creates and returns a new process instance and associates it with the current active process resource on the local computer.

Get a list of all modules that are loaded by a process

The *Modules* property gets a list of all the modules loaded by a process. A module is defined as .dll or .exe file that is loaded into a particular process. Using the *ProcessModule*, you can view information about the module (for example, the name, file name, and memory details).

PerformanceCounter class, PerformanceCounterCategory, and CounterCreationData class

The *PerformanceCounter* class represents the Windows operating system performance counter component. Values written to a *PerformanceCounter* object are stored in the Windows registry. This class allows object-oriented collection and retrieval. Windows has built-in performance counters to measure various resources.

The *PerformanceCounterCategory* class allows you to manage and manipulate *PerformanceCounter* objects and their categories of performance counters. Counters related to the same performance object are grouped into categories that indicate a common focus. Some of the frequent categories used are: Cache, Memory, Objects, PhysicalDisk, Process, Processor, Server, System and Thread.

The *CounterCreationData* class serves as a container object that holds pertinent properties needed to create a *PerformanceCounter* object such as the counter type, counter name and Help.

Start a process both by using and by not using command-line arguments

A process is started by calling the *Start* method of the *Process* class. Since this method is overloaded with five other signatures, a *Process* can be started by passing parameters and/or using the command line. The methods available to start a process and a description on how they are use are displayed below:

- Start() - Start the process using the information in the *StartInfo* property of the *Process* object
- Start(ProcessStartInfo) - Start the process specified by the *ProcessStartInfo* parameter
- Start(String) - Start the process using the name of the document or application file specified by the *String* parameter.
- Start(String, String) - Start the process using the name of the document or application file specified by the *String* parameter and pass a set of command-line arguments specified in the second *String* parameter
- Start(String, String, SecureString, String) - Start the process specifying the name of the application, a username, a password and a domain
- Start(String, String, String, SecureString, String) - Start the process specifying the application name, a set of command-line arguments, a username, a password and a domain

Note: When values specified in the String parameter are separated by spaces, they are considered separate arguments.

StackTrace class

The *StackTrace* class represents a stack trace which is an ordered collection of one or more *StackFrame* objects (the *StackTrace* object can hold up to 512 *StackFrame* objects). This class allows you to view the state of the .NET runtime's call stack at that time to better debug and diagnose application issues and problems. When developing applications, the class will provide more useful information when the application is built using the Debug build configuration because it includes Debug symbols.

StackFrame class

The *StackFrame* class represents a function call on the call stack for the current thread. When a method is called, a *StackFrame* object is added to the stack. Subsequent methods are pushed onto the stack in a last-in, first-out (LIFO) order. Each method is executed and removed from the stack, thus using the *StackTrace* class information regarding the application's state as it processes the methods. The *StackFrame* always includes *MethodBase* information; it may include file name, line number and column number.

4.5 Debug and trace a .NET Framework application by using the System.Diagnostics namespace

Debug class and Debugger class

The *Debug* class provides the methods and properties that programmers can use to debug their code. Public properties exposed by the *Debug* class are:

- AutoFlush - Sets or retrieves a value indicate if the Flush method should be called on the Listeners after each write. This property is static.
- IndentLevel - Sets or retrieves the indent level. This property is static.
- IndentSize - Sets or retrieves the number of spaces to indent. This property is static.
- Listeners - Gets a collection of listeners that monitor the debug output. This property is static.

Some of the public methods exposed by the *Debug* class are:

- Assert - Evaluates a condition and displays a message if the condition is false. This method is static.
- Close -Flushes the output buffer. Calls the Close method on all the attached listener objects. This method is static.
- Fail - Outputs a failure message. This method is static.
- Flush - Flushes the output buffer. Buffered data is written to the Listeners collection. This method is static.
- Indent - Increments the indent level by one (used for formatting). This method is static.
- Print - Writes a message to the trace listeners in the Listeners collection (followed by a line terminator). This method is static.
- ToString - Returns a *String* object that represents the current object
- Unindent - Decrements the indent level by one (used for formatting). This method is static.
- Write - Writes information about the Debug or Trace class listener objects in the Listeners collection. This method is static.
- WriteIf - If a specified condition is met, writes information about the Debug or Trace class listener objects in the Listeners collection. This method is static.
- WriteLine - Writes information (followed by a line terminator) about the Debug or Trace class listener objects in the Listeners collection. This method is static.
- WriteLineIf - If a specified condition is met, writes information (followed by a line terminator) about the Debug or Trace class listener objects in the Listeners collection. This method is static.

Several of the methods provided by the *Debug* class play a more important role for programmers. The *Assert* method is particularly useful in debugging applications when you have code that you expect to always evaluate to *true* or *false*. You can insert *Debug.Assert* statements into your code to trap instances where the code does not evaluate to *true* (or *false*) as expected. This condition will cause the application to automatically break into the debugger. Also useful is the *Fail* method. Unlike the *Assert* method, *Fail* does not use an evaluation to trigger the debugger. It causes the Debugger to break when the method is triggered and renders output regarding the failure.

The *Write*, *WriteIf*, *WriteLine*, and *WriteLineIf* methods are commonly used when debugging by allowing you to send messages to the Output window. The *Write* and *WriteLine* methods send whatever output is passed to the method while the *WriteIf* and *WriteLineIf* only write output if the specified condition is met.

The *Print* method works similarly to the various write methods except it sends the output to attached listener objects.

The *Debugger* class allows you to communicate with the debugger.

The public field exposed by the *Debugger* class is:

- DefaultCategory - The default category of a message with a constant

The public property exposed by the *Debugger* class is:

- IsAttached - Sets or retrieves a value indicating whether a debugger is attached to the process. This property is static.

Some of the public methods exposed by the *Debugger* class are:

- Break - Signals a break to the debugger. This method is static.
- IsLogging - Indicates if the Debugger is currently logging. This method is static.
- Launch - Launches the Debugger and attaches it to a process. This method is static.
- Log - Posts a message to the current debugger. This method is static.
- ToString - Returns a *String* object that represents the current object.

Several of the methods provided by the *Debugger* class play a more important role for programmers. The *Break* method allows programmers to set a breakpoint conditionally in code to signal a breakpoint to the attached *Debugger*. If no *Debugger* is attached when this method is called, users will be prompted to attach a *Debugger*. The *Log* method posts information to listener objects attached to the *Debugger* (if a *Debugger* is present). This method takes three parameters to specify:

- Level – a description of the importance of the message
- Category – the category of the message (this value cannot exceed 256 characters or the value will automatically be truncated to 256 characters)
- Message – The message to log

If no *Debugger* is present, this method does nothing.

Trace class, CorrelationManager class, TraceListener class, TraceSource class, TraceSwitch class, XmlWriterTraceListener class, DelimitedList TraceListener class, and EventlogTraceListener class

The *Trace* class helps trace the execution of the code, allowing you to isolate and correct problems while not interrupting or breaking a running application.

The *CorrelationManager* class (new to the .NET Framework 2.0) correlates trace events that are part of a logical transaction (tagged with a unique identity) generated by a thread.

The *TraceListener* class provides an abstract base class for listeners who monitor trace and debug output. To use the *TraceListener* class, you must have previously enabled tracing or debugging.

The *TraceSource* class (new to the .NET Framework 2.0) enables applications to trace the execution of code and associate the trace messages with their source. Methods provided by the *TraceSource* class provide capabilities to trace events, trace data and issue informative traces. The output can be controlled by the configuration file settings.

The *TraceSwitch* class provides a multi-level *switch* to control tracing and debug output without recompiling code. The *TraceSwitch* class allows you to filter messages and test the level of the switch. The *TraceSwitch* settings can be modified by editing the configuration file settings.

The *XmlWriterTraceListener* class (new to the .NET Framework 2.0) indicates that trace or debugging information should be output as XML-encoded data to a *TextWriter* or *Stream* object. The *XmlWriterTraceListener* can be enabled/disabled using the configuration file settings for the application or a *XmlWriterTraceListener* object can be created directly within your application code.

The *DelimitedListTraceListener* class (new to the .NET Framework 2.0) indicates that trace or debugging information should be output to a *TextWriter* or *Stream* object as delimited text. The actual delimiter is specified by the *Delimiter* property.

The *EventlogTraceListener* class provides a simple listener to direct output from tracing or debugging to be saved to the *EventLog*.

Debugger attributes

DebuggerBrowsableAttribute class
This class (new to the .NET Framework 2.0) determines if (and how) a member is displayed in a debugger variable window. The display states allowed are:

1. Never: the member is not displayed in the data window
2. Collapsed: the member is displayed but collapsed by default
3. RootHidden: the member is not displayed but its constituent objects are (applicable for objects like arrays or collections)

DebuggerDisplayAttribute class
This class (new to the .NET Framework 2.0) determines how a member is displayed in a debugger variable window (for example, inserting captions before variable values, formatting length, or formatting strings).

DebuggerHiddenAttribute class
This class stops a breakpoint from being set inside a specified method or anything that it decorates and is therefore ignored by the debugger.

DebuggerNonUserCodeAttribute class
This class (new to the .NET Framework 2.0) indentifies a type or member that is not part of the user code for the current application and is therefore ignored by the debugger.

DebuggerStepperBoundaryAttribute class
This class (new to the .NET Framework 2.0) indicates that the code following an attribute is to be executed in run mode (not step mode). This class is relevant for code executing in the boundaries of the *DebuggerNonUserCodeAttribute*.

DebuggerStepThroughAttribute class
This class tells the debugger that the indicated code should be stepped over and not display in the debugging windows. You can still set a breakpoint in a method marked by this class.

DebuggerTypeProxyAttribute class
This class (new to the .NET Framework 2.0) specifies how a given type is displayed.

DebuggerVisualizerAttribute class
This class (new to the .NET Framework 2.0) specifies the visualizer for a given class. Support for visualizers depends on the host debugger. Support for this capability was first available in Visual Studio 2005.

4.6 Embed management information and events into a .NET Framework application

System.Management Namespace

The *System.Management* namespace provides management information and events about the current system, devices and applications via the Windows Management Instrumentation (WMI) technology.

Retrieve a collection of Management objects by using the ManagementObjectSearcher class and its derived classes.

The *ManagementObjectSearcher* class retrieves a collection of *ManagementObjects* based on a WMI query. This class provides the following capabilities:

- Enumerate all disk drivers, network adapters, and processes on a computer
- Retrieve information about all network connections
- Retrieve information about all services that are paused

ManagementQuery class
The *ManagmentQuery* class is a base class for all WMI queries (providing an abstract class for all management query objects).

EventQuery class
The *EventQuery* class represents a query object used to query WMI event query objects.

ObjectQuery class
The *ObjectQuery* class represents a query object used to query instances and classes.

Subscribe to management events by using the ManagementEvent Watcher class.

The *ManagementEventWatcher* class allows you to subscribe to temporary event notifications based on a specific event query within the WMI. The steps to use the *ManagementEventWatcher* class are:

- Instantiate a handle on a new *ManagementEventWatcher* object
- Associate the object to an *EventQuery* object
- Call the *WaitForNextEvent* method. This method will wait for the event specified by the query and return it once it arrives.
- Stop the notifications by calling the *Stop* method to cancel the subscription.

Implementing serialization and input/output functionality in a .NET Framework application

5.1 Serialize or deserialize an object or an object graph by using runtime serialization techniques.

System.Runtime.Serialization Namespace

Objects are serialized and deserialized so they they can be stored or transferred, then later re-created. Serializing converts an object (or objects) into a linear sequence of bytes to be stored or transferred; deserializing converts a previously serialized sequence of bytes back into an object.

Serialization interfaces

1. IDeserializationCallback interface - indicates that a class is to be notified when the entire object has been deserialized
2. IFormatter interface - provides functionality for formatting serialized objects. This interface is implemented by any class identified as a formatter in the *Serialization* class.
3. IFormatterConverter interface - the base implementation of the *IFormatterConverter* interface which uses the *Convert* class and the *IConvertible* interface
4. ISerializable interface - permits an object to control its own serialization and deserialization

Serilization attributes (all new to the .NET Framework 2.0):

1. OnDeserializedAttribute class –the method is called immediately after deserialization of the object
2. OnDeserializingAttribute class –the method is called during the deserialization of the object
3. OnSerializedAttribute class –the method is called immediately after serialization of the object
4. OnSerializingAttribute class –the method is called during the serialization of the object
5. OptionalFieldAttribute class - specifies that a field can be missing from a serialization stream preventing *BinaryFormatter* and *SoapFormatter* from throwing an exception

SerializationEntry structure and SerializationInfo class

The *SerializationEntry* structure holds the value, Type object and name of a serialized object.
The *SerializationInfo* class stores all the data needed to serialize or deserialize an object.

ObjectManager class

The *ObjectManager* class keeps track of objects as they are deserialized.

Formatter class, FormatterConverter class, and FormatterServices class

The *Formatter* class implements the base functionality for the CLR serialization formatters (though this class is not CLS compliant). The *Formatter* class is an abstract base class for all runtime serialization formatters. It provides some helper methods for implementing the *IFormatter* interface.

The *FormatterConverter* class implements the base implementation of the *IFormatterConverter* interface that uses the *Convert* class and the *IConvertible* interface.

The *FormatterServices* class provides static methods to help implement a Formatter for serialization.

StreamingContext structure

The *StreamingContext* structure provides the source and destination bits of a given serialized stream that a formatter is using.

5.2 Control the serialization of an object into XML format by using the System.Xml.Serialization namespace.

Serialize and deserialize objects into XML format by using the XmlSerializer class

The .NET Framework 2.0 provides several libraries for reading and writing XML files. The *System.Xml.Serialization* namespace provides methods for converting objects to and from XML-formatted documents and streams.

To create a class that can be serialized using XML serialization:

1. Specify the class as public
2. Specify all members that are to be serialized as public (private and protected members will be skipped during serialization)
3. Create a constructor with no parameters

The *System.Xml.Serialization* class provides several *Attribute* classes used to control the serialization. These attributes (*XMLArray*, *XMLAttribute*, *ZMLElement*, *XMRoot*, *XMLText*, *XMLType*) can make a serialized class conform to a specific XML requirement or schema.

You can implement XML Serialization interfaces to provide custom formatting for XML serialization. The *IXmlSerializable* Interface provides custom formatting for XML serialization and deserialization. This interface can help control how your object is serialized or deserialized by the XmlSerializer or control the schema.

Several delegates and event handlers are provided by the System.Xml.Serialization namespace to handle events and provide more control over the XML serialization. For example, *XmlAttributeEventHandler* provides a method that handles the UnknownAttribute , XmlElementEventHandler provides a method that handles the UnknownElement event of an XmlSerializer, and XmlNodeEventHandler provides a method that handles the UnknownNode event of an XmlSerializer.

5.3 Implement custom serialization formatting by using the Serialization Formatter classes.

The *System.Runtime.Serialization* namespace provides two methods for formatting serialized data, *SoapFormatter* and *BinaryFormatter*.

SoapFormatter class

The *SoapFormatter* class serializes or deserializes an object, or a graph of connected objects, in SOAP format. The SOAP format is XML-based, and a reliable way to serialize objects transmitted and read by non .NET framework applications.

BinaryFormatter class

The *BinaryFormatter* class serializes or deserializes an object, or a graph of connected objects, in binary format. For objects that will be serialized and deserialized by .NET Framework-based applications, this formatter is an efficient mechanism.

5.4 Access files and folders by using the File System classes.

System.IO Namespace

The *System.IO* namespace provides types and members that allow reading and writing to files and data streams as well as basic files and directory support. Essentially, the classes of the *System.IO* namespace allow you to navigate and manipulate files, directories and drives. The two types of classes are:

1. Informational – exposes all the system information about the file system objects (files, directories and drives).
2. Utilities – provides methods to perform operations on the file system objects

File class and FileInfo class

The *File* class provides methods to manipulate files (create, copy, delete, move, rename, append and open) and a mechanism for creating *FileStream* objects. The *File* class can also be used to manipulate file attributes (such as *DateTime* information related to the creation, access and writing of a file).

Note: the methods of the *File* object always perform security checks when performing operations on all methods. By default, full read/write access is granted to all users for new files.

The *FileInfo* class provides methods for copying, moving, renaming, creating, opening, deleting, and appending to files as well as the creation of *FileStream* objects.

Directory class and DirectoryInfo class

The *Directory* class provides methods for copying, renaming, creating, moving, deleting, and enumerating through directories and subdirectories. The *Directory* class can also manipulate *DateTime* attributes (such as information related to the creation, access and writing of a directory).

The *DirectoryInfo* class allows you to copy, rename, create, move, delete and enumerate through directories and subdirectories, and access and manipulate a single directory on the file system.

DriveInfo class and DriveType enumeration

The *DriveInfo* class (new to the .NET Framework 2.0) provides programmatic access to information on a drive such as the drive type, drive capacity, free space and availability

The *DriveType* enumeration (new to the .NET Framework 2.0) is used by the *DriveInfo* class to indicate the drive type. This class defines the following constants for drive types: CDRom, Fixed, Network, NoRootDirectory, Ram, Removable, and Unknown .

FileSystemInfo class and FileSystemWatcher class

The *FileSystemInfo* object can be a file or directory; it is the base class for the *FileInfo* and *DirectoryInfo* classes.

Note: When the *FileSystemInfo* object is first instantiated, the *Refresh* method is automatically called and cached information on APIs attributes is returned. Thereafter, you must manually call the *Refresh* method to get updated attribute information.

The *FileSystemWatcher* class provides methods for monitoring file system directories for modifications. This class listens to the file system (on a local computer, network drive or remote computer) for change notifications and will raise an event if a directory, subdirectory or file in a directory is modified.

You can use the *Filter* property to watch a specific file, or set it to an empty string ("") or a wildcard ("*.*") to watch for changes in all files. Other considerations when using the *FileSystemWatcher* class are:

- Hidden files are not ignored
- Changes to files may be reported using the short 8.3 file name format on some systems
- The FileSystemWatcher class contains a link demand and an inheritance demand at the class level that applies to all members. A *SecurityException* will be thrown if the immediate caller or the derived class does not have full-trust permission.

Path class

The *Path* class provides methods to manipulate *String* instances containing the file or directory path information in a cross-platform manner.

ErrorEventArgs class and ErrorEventHandler delegate

The *ErrorEventArgs* class contains the *Exception* that caused the error event. The *GetException* method is called to retrieve the exception. The *ErrorEventHandler* delegate is the method that will handle the error event of the *FileSystemWatcher* object. When the *ErrorEventHandler* delegate is created, the method you want to handle the event is specified. This method is called whenever the error is found unless the delegate is removed.

RenamedEventArgs class and RenamedEventHandler delegate

The *RenamedEventArgs* class provides data for the *Renamed* event. This class extends the *FileSystemEventArgs* class by adding an old name field and an old full path field to specify the previous name and full path of the affected file or directory.

The *RenamedEventHandler* delegate represents the method that handles the *Renamed* event of a *FileSystemWatcher* class. When the *RenamedEventHandler* delegate is created, the method that you want to handle the event is specified. This method is called whenever the error is found unless the delegate is removed.

5.5 Manage byte streams by using Stream classes.

FileStream class

The *FileStream* class provides the functionality to open file streams for reading, writing, opening and closing. The *FileStream* class exposes the *Stream* of a file supporting synchronous and asynchronous read and write operations. It can also manipulate file-related operating system handles such as pipes, standard input and standard output. It buffers input and output for better performance.

Note: If the *FileStream* object does not have an exclusive hold on its handle, another thread could access the file handle concurrently and change the position of the operating system's file pointer. If the system's handle position has changed, the *FileStream* object will read the stream from the file again which could negatively affect performance.

Stream class

The *Stream* class represents a generic view of a sequence of bytes. Streams are a common type of reference type that provide the programmatic ability to read and write to the disk and communicate across a network. The *System.IO.Stream* type is the base type for all subsequent stream type tasks. For example, the *StreamReader* class enables developers to read from text files and the *streamWriter* class enables developers to write to text files. Streams involve three basic operations:

1. Reading from *Streams*
2. Writing to *Streams*
3. Seeking within a *Streams* (querying and modifying of the current position within a stream)

MemoryStream class

The *MemoryStream* class allows you to create *Streams* that use memory as a backing store to hold the stream rather than the disk or network connection. When the *MemoryStream* object is created, the data is encapsulated and stored as an unsigned byte array (which can be created as empty).

BufferedStream class

The *BufferedStream* class provides a buffering layer to read and write operations to another *Stream*.

The buffer is a block of bytes stored in memory that cache data, thus reducing the number of calls to the operating system and improving performance. Once the buffer is flushed, the data is written to the underlying stream.

5.6 Manage the .NET Framework application data by using Reader and Writer classes.

StringReader class and StringWriter class

The *StringReader* class implements a *TextReader* object that reads from a string.

The *StringWriter* class implements a *TextWriter* object for writing information to a string. The information is stored in an underlying *StringBuilder*.

TextReader class and TextWriter class

The *TextReader* class provides a reader that can read a sequential series of characters. The *TextReader* class is an abstract base class of *StreamReader* and *StringReader* classes used to open a text file for reading a stream or specified range of characters.

The *TextWriter* class provides a reader that can write a sequential series of characters. The *TextReader* class is an abstract base class of *StreamWriter* and *StringWriter* classes used to open a text file for writing a stream or specified range of characters. A *TextWriter* object allows you to write an object to a string, write strings to a file or serialize XML.

StreamReader class and StreamWriter class

The *StreamReader* class provides the ability to read characters from a stream as a string by implementing a *TextReader* object (which reads characters from a byte stream in a particular encoding). This class can read lines of characters from a standard text file.

The *StreamWriter* class allows characters to be written from a stream as a string by implementing a *Text-Writer* object (which writes characters from a byte stream in a particular encoding).
Note: *StreamReader* and *StreamWriter* default to using an instance of UTF8Encoding unless specified otherwise

BinaryReader class and BinaryWriter class

The *BinaryReader* class allows primitive data types to be read as binary values in a specific encoding.
The *BinaryWriter* class allows primitive data types to be written as binary values in a specific encoding.

Note: When the *BinaryReader* and *BinaryWriter* signatures specify a *Stream* as a parameter, they default to using an instance of UTF8Encoding unless specified otherwise

5.7 Compress or decompress stream information in a .NET Framework application (refer System.IO.Compression namespace), and improve the security of application data by using isolated storage.

The .NET Framework contains two methods for compressing data using standard compression algorithms: GZIP and DEFLATE. Both compression algorithms have no patent protection and allow the compression or decompression of data up to 4GB. The compression stream methods exposed to use compression/decompression are *GZIPStream* and *DeflateStream*. Both methods use the same algorithm for compressing and decompressing data. However, the *GZIPStream* class specification includes additional information about the compression which may be required by other tools outside. Therefore, if building entirely self-contained applications, you can use the *DeflateStream* class (which is slightly smaller) but if the data might be shared outside the application, you should use the *GZIPStream* class.

The *IsolatedStorage* class gives applications a safe way to store information and files while not granting users access to specific files of folders on the file system. Isolated stores are scoped to a particular assembly:

- Assembly/Machine – creates the store to keep information specific to the calling assembly and the local machine (useful when creating application level data)
- Assembly/User – creates the store to keep information specific to the calling assembly and the current user (useful when creating user level data)

IsolatedStorageFile class

The *IsolatedStorageFile* class provides the base functionality to create files and folders in the isolated storage environment without having to specify a particular path within the file system.

IsolatedStorageFileStream class

The *IsolatedStorageFileStream* class encapsulates the stream used to create files in the isolated storage environment and exposed the file. This allows you to read, write and create files in isolated storage.

DeflateStream class

The *DeflateStream* class (new to the .NET Framework 2.0) allows data to be compressed through another stream using the Deflate compression method. The Deflate algorithm is a standard algorithm for lossless file compression and decompression. The Deflate format includes a cyclic redundancy check value for detecting data corruption.

GZipStream class

The *GZipStream* class (new to the .NET Framework 2.0) allows data to be compressed through another stream using the GZIP method. The GZIP format includes a cyclic redundancy check value for detecting data corruption. The data can be be extended to use other compression formats

Improving the security of the .NET Framework applications by using the .NET Framework 2.0 security features

6.1 Implement code access security to improve the security of a .NET Framework application.

System.Security Namespace

The *System.Security* namespace provides the fundamental structure for the CLR security system and permissions management. Some of the more common classes contained within this namespaces are outlined in this chapter.

SecurityManager class
The *SecurityManager* class provides the primary access mechanism for classes that interact with the security system in the .NET Framework. You do not create new instances of the *SecurityManager* class. You use this class to utilize methods to access and manipulate security policy configuration. This class' methods let you check execution rights, enable/disable security, manage caller permission and manage security.

CodeAccessPermission class
The *CodeAccessPermission* class provides the underlying structure for all code access permissions. It uses a stack walk to verify permission of the callers of the code (represented as "growing down" where methods higher in the call stack call methods lower in the call stack).

Code Access security policies can be modified at the computer, user and enterprise policy levels by using the Code Access Security Policy tool (Caspol.exe). When running Capspol.exe from the command line, enter "Caspol -?" for a complete set of options and instructions. Using the caspol tool, you can add code groups, change code group permissions, list and modify code groups for a specified policy, manage machine level policies, manage enterprise-level and user-level policies, manage assemblies that implement custom security, and disable/enable security checks.

PermissionSet class and NamedPermissionSet class
The *PermissionSet* class represents a collection of permissions. Using a *PermissionSet* object, you can manage several different permissions as a group (add, removed, deny, set and evaluate permissions). Permissions in the CLR are objects that describe a set of operations that can be secured for specific resources. Permissions are used by application code and the .NET Framework; code can request permission to run, the security system policy can allow code to run, code can demand that calling code has permission, and code can override the security stack using assert/deny/permit only functions.

Standard Security interfaces

- IEvidenceFactory interface - returns a handle to an object's *Evidence* object. The *Evidence* class is a collection that holds sets of objects representing information that constitutes te input to security policy decisions, such as code signatures, code origins and objects of any type. Specifically, this class holds two types of *Evidence* classes:
- *Host evidence - provides evidence regarding the origin of code, signatures, etc.*
- Assembly evidence - part of the assembly; allows developers and administrators to attach custom evidence to the assembly.
- IPermission interface – defines the methods implemented by permission types. When writing a new permission, programmers must implement the *IPermission* interface into the class.

6.2 Implement access control by using the System.Security. AccessControl classes.

The *System.Security.AccessControl* namespace (new to the .NET Framework 2.0) provides programming elements that manage/audit access to security-related actions on securable objects. Some of the classes provided by this namespace are described below.

DirectorySecurity class, FileSecurity class, FileSystemSecurity class, and RegistrySecurity class

The *DirectorySecurity* class (new to the .NET Framework 2.0) represents access rights for a system directory and how access attempts are audited. The access and audit rights are represented as a set of rules. Each rule is represented by a *FileSystemAccessRule* object while each audit rule is represented by a *FileSystemAuditRule* object.

The *FileSecurity* class (new to the .NET Framework 2.0) represents the access rights and audit security for a specific file. As with the *DirectorySecurity* class, access and audit rights are represented as a set of rules (access rules are represented by a *FileSystemAccessRule* object while each audit rule is represented by a *FileSystemAuditRule* object).

The *FileSystemSecurity* class (new to the .NET Framework 2.0) represents the access rights and audit security for a specific file or directory. The *FileSystemSecurity* class is the base class for the FileSecurity and DirectorySecurity classes; represents all the access rights for a system file or directory, and defines how access attempts are audited. As with the previous two classes, access and audit rights are represented as a set of rules.

The *RegistrySecurity* class (new to.NET Framework 2.0) represents the access rights and audit security for a registry key with access and audit rights represented as a set of rules.

AccessRule class (new to the .NET Framework 2.0 represents
The *AccessRule* represents a user's identity, an access mask, an access control type and indicates how rules are inherited by child objects and how their inheritance is propagated.

AuthorizationRule class and AuthorizationRuleCollection class
The *AuthorizationRule* (new to the .NET Framework 2.0) allows you to manage access to securable objects. Its derived classes of *AccessRule* and *AuditRule* offer specializations for access and audit functionality. The *AuthorizationRuleCollection* (new to the .NET Framework 2.0) represents a collection of *AuthorizationRule* objects.

CommonAce class, CommonAcl class, CompoundAce class, GeneralAce class, and GenericAcl class
The *CommonAce* class (new to the .NET Framework 2.0) is an access control entry.

The *CommonAcl* class (new to the .NET Framework 2.0) is an access control list (ACL). It is the base class for the *DiscretionaryAcl* and *SystemAcl* class.

The *CompoundAce* class (new to the .NET Framework 2.0) represents a compound Access Control Entry (ACE).

The *GenericAcl* class (new to the .NET Framework 2.0) represents the access control list (ACL). It is the base class for *CommonAcl*, *DiscretionaryAcl*, *RawAcl* and *SystemAcl*.

AuditRule class
The *AuditRule* class (new to the .NET Framework 2.0) provides access to the user's identity and an access mask; contains information about how rules are inherited by child objects, how inheritance is propagated, and for what conditions it is to be audited.

MutexSecurity class, ObjectSecurity class, and SemaphoreSecurity class (all new to the .NET Framework 2.0)
The *MutexSecurity* class provides access rights to a named system mutex and specifies how access attempts are audited. Access rights are expressed as rules (access rules are represented by a *MutexAccessRule* and auditing rules by a *MutexAuditRule* object). The *ObjectSecurity* class provides control access to objects without direct manipulation of Access Control Lists (ACLs) and is the abstract base class for the *CommonObjectSecurity* and *DirectoryObjectSecurity* classes. The *SemaphoreSecurity* class provides access rights to a named system semaphore and specifies how access attempts are audited. Access rights are expressed as rules (access rules are represented by a *SemaphoreAccessRule* and auditing rules are represented by a *SempahoreAuditRule* object).

6.3 Implement a custom authentication scheme by using the System.Security.Authentication classes.

The *System.Security.Authentication* namespace (new to the .NET Framework 2.0) provides a set of enumerations (*CipherAlgorithmType*, *ExchangeAlgorithmType*, *HashAlgorithmType*, and *SslProtocolType*) which describe the security of an authenticated connection. These enumerations are:

- CipherAlgorithmType - the possible cipher algorithms for the *SslStream* class
- *ExchangeAlgorithmType - the algorithm that creates keys shared by the client and server*
- *HashAlgorithmType - the algorithm that generates message authentication codes (MACs)*
- SslProtocolType - the possible versions of *SslProtocols*

6.4 Encrypt, decrypt, and hash data by using the System.Security.Cryptography classes.

The *System.Security.Cryptography* namespace includes cryptographic services: secure data encoding/decoding, hashing, random number generation, message authentication, and other methods. Detailing the many classes, interfaces, structures and enumerations of the *System.Security.Cryptography* namespace is outside the scope of this book, buta list of the common classes has been provided below. You should review it before the exam:

- DES class and DESCryptoServiceProvider class
- HashAlgorithm class
- DSA class and DSACryptoServiceProvider class
- SHA1 class and SHA1CryptoServiceProvider class
- TripleDES and TripleDESCryptoServiceProvider class
- MD5 class and MD5CryptoServiceProvider class

- RSA class and RSACryptoServiceProvider class
- RandomNumberGenerator class
- CryptoStream class
- CryptoConfig class
- RC2 class and RC2CryptoServiceProvider class
- AssymetricAlgorithm class
- ProtectedData class and ProtectedMemory class
- RijndaelManaged class and RijndaelManagedTransform class
- CspParameters class
- CryptoAPITransform class
- Hash-based Message Authentication Code (HMAC) classes (HMACMD5 class , HMACRIPEMD160 class , HMACSHA1 class , HMACSHA256 class , HMACSHA384 class , HMACSHA512 class)

6.5 Control permissions for resources by using the System. Security.Permissions classes.

The *System.Security.Permissions* namespace provides access to classes that control access to operations and resources based on policies. Detailing the many classes, interfaces, and enumerations of the *System. Security. Permissions* namespace is outside the scope of this book, but the common classes are listed below. You should review them before the exam:

- SecurityPermission class - a set of security permissions applied to code
- PrincipalPermission class - permits checks against the active principal using the language constructs defined for both declarative and imperative security actions
- FileIOPermission class - manages the ability to access files and folders
- StrongNameIdentityPermission class - outlines the identity permission for strong names
- UIPermission class - manages the permissions related to user interfaces and the clipboard
- UrlIdentityPermission class - outlines the identity permission for the URL from which the code originates
- PublisherIdentityPermission class - represents the identity of a software publisher
- GacIdentityPermission class - manages the identity permission for files originating in the global assembly cache. This class is new to the .NET Framework 2.0.
- FileDialogPermission class - manages the ability to access files or folders through a file dialog
- DataProtectionPermission class - manages the ability to access encrypted data and memory. This class is new to the .NET Framework 2.0.
- EnvironmentPermission class - manages access to system and user environment variables
- IUnrestrictedPermission interface - allows a permission to expose an unrestricted state

- RegistryPermission class - manages the ability to access registry variables
- IsolatedStorageFilePermission class - indicates the allowed use of a private virtual file system
- KeyContainerPermission class - manages the ability to access key containers. This class is new to the .NET Framework 2.0.
- ReflectionPermission class - manages access to metadata through the *System.Reflection* APIs
- StorePermission class - outlines the identity permission for strong names. This class is new to the .NET Framework 2.0.
- SiteIdentityPermission class - outlines the identity permission for the Web site from which the code originates
- ZoneIdentityPermission class - outlines the identity permission for the zone from which the code originates

Control code privileges by using System.Security.Policy classes.

The System.Security.Policy namespace contains code groups, membership conditions, and evidence classes used to create rules applied by CLR security policy system.

ApplicationSecurityInfo class and ApplicationSecurityManager class
The *ApplicationSecurityInfo* class (new to.NET Framework 2.0) contains security evidence for an application by providing security data regarding the manifest-activated application.

The *ApplicationSecurityManager* class (new to.NET Framework 2.0) manages trust decisions and provides essential information for executing manifest activated applications.

ApplicationTrust class and ApplicationTrustCollection class
The *ApplicationTrust* class (new to.NET Framework 2.0) contains security decisions about an application.

The *ApplicationTrust* object is instantiated using the *DetermineApplicationTrust* method of the trust manager.

The *ApplicationTrustCollection* class (new to the .NET Framework 2.0) simply represents a collection of *ApplicationTrust* objects.

Evidence class and PermissionRequestEvidence class
The *Evidence* class provides input to security policy decisions (signatures, location of origin of code, any object of any type recognized by security policy as evidence). As previously described, the *Evidence* class is a collection holding a set of objects (host evidence and assembly evidence).

The *PermissionRequestEvidence* class outlines evidence representing permission requests (minimum permissions the code needs to run, permissions code can use if granted, and permissions the code explicitly asks not to be granted).

CodeGroup class, FileCodeGroup class, FirstMatchCodeGroup class, NetCodeGroup class, and UnionCodeGroup class

The *CodeGroup* class represents the abstract base class from which all implementations of code groups (the building blocks of code access security policy) must derive. Each policy level contains a root code group which can have child code groups. Each child code group can have its own child code groups. Each of these child groups has subsequent code groups, and so on, to any number of levels, creating a code group tree. Each code group has a membership condition that determines if an assembly belongs to it based on its evidence.

Condition classes

1. AllMembershipCondition class - represents a membership condition that matches all code (typically used on the root code group of a policy level, therefore, the policy applies to all code).
2. ApplicationDirectory class - provides the application directory as evidence for policy evaluation.
3. ApplicationDirectoryMembershipCondition class – indicates if an assembly belongs to a code group by testing its application directory.
4. GacInstalled class - confirms that a code assembly originates in the global assembly cache (GAC) as evidence for policy evaluation. This class is new to the .NET Framework 2.0.
5. GacMembershipCondition class - indicates is an assembly belongs to a code group by testing its global assembly cache membership. All assemblies installed in the global assembly cache are granted the FullTrust permission set. This class is new to the .NET Framework 2.0.
6. Hash class - supplies evidence about the hash value for an assembly as a unique value that corresponds to a particular set of bytes. This eliminates all ambiguity since the hash value designates the assembly as a set of bytes rather than referring to an assembly by name, version, or other designation. Also, hash values are a very cryptographically secure way to refer to specific assemblies in policy without having to use digital signatures.
7. HashMembershipCondition class - indicates whether an assembly belongs to a code group by testing the hash value.
8. Publisher class - Provides the Authenticode X.509v3 digital signature of a code assembly as evidence for policy evaluation Determines whether an assembly belongs to a code group by testing its software publisher's Authenticode X.509v3 certificate.
9. PublisherMembershipCondition class - indicates whether an assembly belongs to a code group by testing its software publisher's Authenticode X.509v3 certificate.
10. Site class - supplies the Web site from which a code assembly originates as evidence for policy evaluation.
11. SiteMembershipCondition class - indicates is an assembly belongs to a code group by testing the site from which it originated.
12. StrongName class - supplies the strong name of a code assembly as evidence for policy evaluation.
13. StrongNameMembershipCondition class – indicates if an assembly belongs to a code group by testing its strong name.
14. Url class –supplies the URL from which a code assembly originates as evidence for policy evaluation
15. UrlMembershipConditon class – indicates if an assembly belongs to a code group by testing its URL.
16. Zone class - supplies the security zone of a code assembly as evidence for policy evaluation.
17. ZoneMembershipCondition class – indicates if an assembly belongs to a code group by testing its zone of origin.

PolicyLevel class and PolicyStatement class
The *PolicyLevel* class represents the security policy levels for the common language runtime. The highest level of security policy is enterprise-wide. Successive lower levels of hierarchy represent further policy restrictions, but can never grant more permissions than allowed by higher levels.
The following policy levels are implemented:

1. Enterprise: security policy for all managed code in an enterprise.
2. Machine: security policy for all managed code run on the computer.
3. User: security policy for all managed code run by the user.
4. Application domain: security policy for all managed code in an application.

The *PolicyStatement* class represents the statement of a *CodeGroup* describing the permissions and other information that apply to code with a particular set of evidence. A *PolicyStatement* consists of a set of granted permissions, and possible special attributes for the code group.

IApplicationTrustManager interface and IMembershipCondition interface
The *IApplicationTrustManager* interface (new to the .NET Framework 2.0) determines whether an application should be executed and, if so, which set of permissions should be granted to the application.

The *IMembershipCondition* interface represents the test to determine if a code assembly is a member of a code group.

6.7 Access and modify identity information by using the System.Security.Principal classes.

The *System.Security.Principal* namespace defines the principal object representing the security context under which code is running. In other words, it contains information about the identity and roles(s) that the current code (or user) is associated with and provides the ability to check the role membership of the current user. Specifically, when handling requests requiring authorization, the .NET runtime will examine the Principle attached to the current thread in order to determine the identity and roles of a user. The *System.Security.Principal* namespace contains two classes, the *GenericPrincipal and WindowsPrincipal* class, used to manage the properties of the *System.Security.Principal* object. The *IPrincipal* interface allows you to define your own properties. The classes and interface are outlined below.

GenericIdentity class and GenericPrincipal class
The *GenericIdentity* class simply represents a generic user on whose behalf the code is executing.

The *GenericPrinciple* class represents the roles of the current user.

WindowsIdentity class and WindowsPrincipal class
The *WindowsIdentity* class simply represents the identity of the current user. You can instantiate a handle on the *WindowsIdentity* object by calling the *GetCurrent* method of *WindowsIdentity* class.

The *WindowsPrinciple* class provides the ability to programmatically check the Windows group membership of a Windows user.

NTAccount class and SecurityIdentifier class
The *NTAccount* class (new to the .NET Framework 2.0) represents a user or group account. The *Value* property can be used to return an uppercase string representation of the NTAccount object. The *ToString* method returns the account name in the familiar "Domain\Account" format.

The *SecurityIdentifier* class (new to the .NET Framework 2.0) represents a security identifier (SID) and provides marshaling and comparison operations for SIDs. The *Value* property returns the Security Descriptor Definition Language (SDDL) string in uppercase for the security identifier (SID) represented by the *SecurityIdentifier* object. Other methods can be used to determine if the security identifier is a valid Windows account SID, is in the same domain as a specified SID, matched a well known SID type, etc.

IIdentity interface and IPrincipal interface

The *IIdentity* interface provides basic functionality of an identity object (the user on whose behalf the code is running). The *IIdentity* interface is limited to three properties: *AuthenticationType* which gets the type of authentication used, *IsAuthenticated* which gets a value that indicates whether the user has been authenticated, and *Name* which returns the name of the current user.

The *IPrincipal* interface provides the basic functionality of a principal object, which has been described in great detail throughout this chapter. The principal object represents the security context of the user on whose behalf the code is running (the user's identity (*IIdentity*) and any roles to which the user belongs). All principal objects are required to implement the *IPrincipal* interface. The *IPrincipal* interface has one property, *Identity* which returns the identity of the current user and one method, *IsInRole* which indicates if the current principal belongs to the specified role.

WindowsImpersonationContext class

The *WindowsImpersonationContext* class represents the Windows user prior to an impersonation operation by revert back a user's previous identity after the user impersonates another user. The *Undo* method reverts the user context to the Windows user represented by the *WindowsImpersonationContext* object.

Note: Microsoft Windows 98 and Windows Millennium Edition platforms do not have users or user tokens. Therefore, impersonation cannot take place on those platforms.

IdentityReference class and IdentityReferenceCollection class

The *IdentityReference* class (new to the .NET Framework 2.0) represents an identity. This is the base class for the *NTAccount* and *SecurityIdentifier* classes. The *Value* property returns the string representation of the identity represented by the *IdentityReference* object.

The *IdentityReferenceCollection* class (new to the .NET Framework 2.0) represents a collection of *IdentityReference* objects. This class provides the ability to convert sets of *IdentityReference* derived objects to *IdentityReference* derived types. One of the methods of this class, *Translate*, converts the objects in the *IdentityReferenceCollection* collection to the specified type.

Implementing interoperability, reflection, and mailing functionality in a .NET Framework application

7.1 Expose COM components to the .NET Framework and the .NET Framework components to COM.

System.Runtime.InteropServices namespace

Interoperation refers to the process of interacting with unmanaged code from within managed code (the .NET environment). The *System.Runtime.InteropServices* namespace provides a wide variety of members that support COM interop and platform invoke services

Import a type library as an assembly.

You can add references to type libraries from within Visual Studio 2005 by selecting Project, Add Reference from the drop-down menu. The first two tabs will display options for .NET assemblies and COM object references.

The Type Library Importer (Tlbimp.exe) tool used to import COM components from the Visual Studio 2005 command prompt. Using this tool creates a new .NET assembly that is available from the .NET tab when using the "Add Reference" dialog box.

The format for this tool is:

tlbimp <dllname>.dll

or to change the name of the dll use:

tlbimp <dllname>.dll /out:<desiredname>.dll

Alternatively, you can also use Visual Studio 2005 or the *System.Runtime.InteropServices* namespace. Prior to using this tool, the COM object (dll file) must first be registered, for example using Regsvr32 from the command line.

TypeConverter class
The *TypeConverter* class provides a way of to convert types of values to other data types and provides a way to access standard values and subproperties. The *TypeConverter* can be used for string-to-value conversions or translation to or from supported data types at design time as well as run time. Most native types already have default type converters built into them to allow string-to-value conversions and perform validation checks.

While it is outside the scope of this book, you should become familiar with performing the following tasks using Visual Studio 2005:

- Creating COM types in managed code
- Compiling an interop project
- Deploying an interop application

- Qualifying the .NET Framework types for interoperation

When building or deploying COM enabled assemblies, the following guidelines should be followed:

- All classes must use a default constructor with no parameters
- Any exposed type must be public
- Any exposed member must be public
- Abstract classes will be able to be consumed
- Applying Interop attributes, such as the *ComVisibleAttribute* class

The *ComVisibleAttribute* class manages the accessibility of an individual managed type or member (or of all types within an assembly) to COM. This attribute can be applied to assemblies, interfaces, classes, structures, delegates, enumerations, fields, properties, or methods.

- Packaging an assembly for COM
- Deploying an application for COM access

7.2 Call unmanaged DLL functions in a .NET Framework application, and control the marshaling of data in a .NET Framework application.

Platform Invoke

Platform Invoke (or P/Invoke) allows programmers to call an unmanaged Windows API. The platform Invoke is managed from the *System.Runtime.InteropServices* namespace. To use the platform Invoke functionality, you simply do the following:

1. Create a new static or shared external method
2. Decorate the new method with the *DLLImport* attribute and specify the library to call
3. Call the method from your code.

Create a class to hold DLL functions

Another option rather than using Platform Invoke is to create a class to contain the DLL functions you wish to use. This method provides the following advantages: the code appears to developers using the class as other .NET code contained within the application, developers are not required to remember API specific references, it is more consistent and less error prone.

While outside the scope of this book, in preparation for the exam you should become familiar with how to call a DLL function as well as call a DLL function in special cases, such as passing structures and implementing callback functions.

The DllImportAttribute class
The *DllImportAttribute* class indicates that the attributed method is exposed by an unmanaged dynamic-link library (DLL) as a static entry point. This attribute can only be applied to methods but provides information needed to call a function exported from an unmanaged DLL.

Default marshaling behavior
Marshaling determines how data is passed in method arguments and return values between managed and unmanaged memory during calls (performed by the CLR's marshaling service as a run-time activity). The CLR provides two mechanisms for interoperating with unmanaged code, platform Invoke and COM interop.

- Marshal data with Platform Invoke - enables managed code to call functions exported from an unmanaged library
- Marshal data with COM Interop - enables managed code to interact with COM objects through interfaces

Both types of marshaling move method arguments between caller and callee and back again (when required). With platform Invoke, invoke calls can flow only from managed to unmanaged code while method calls can flow in either direction with COM Interop (of course, data can flow in both directions as In or Out parameters with either method).

MarshalAsAttribute class and Marshal class
The *Marshal* class provides methods for allocating unmanaged memory, copying unmanaged memory blocks, and converting managed to unmanaged types, as well as other miscellaneous methods used when interacting with unmanaged code.

The *MarshalAsAttribute* class determines how to marshal the data between managed and unmanaged code. This attribute can be applied to parameters, fields, or return values. The *MarshalAsAttribute* attribute is optional and is only necessary when a given type can be marshaled to multiple types. Also, each data type has a default marshaling behavior.

7.3 Implement reflection functionality in a .NET Framework application (refer System.Reflection namespace), and create metadata, Microsoft intermediate language (MSIL), and a PE file by using the System.Reflection.Emit namespace.

SystemReflection namespace

The *SystemReflection* namespace contains types that retrieve information about assemblies, modules, members, parameters, and other entities in managed code by examining their metadata. This namespace provides the capability to interrogate types in the type system and generate code on the fly.

Note: Metadata is the information the CLR uses to load and execute code. It includes type information (information about methods, properties, events, delegates, and enumerations) for all classes, delegates, and interfaces in the assembly.

Assembly class
The *Assembly* class defines an assembly. An assembly is defined as a "reusable, versionable, and self-describing building block of a common language runtime application". The assembly essentially provides the infrastructure to allow the runtime to fully understand the contents of an application. It also enforces any versioning and dependency rules defined by the application.

Assembly attributes

Assembly attributes provide additional information about the assembly. You can retrieve assembly information using the *GetCustomAttributes* method of the *Assembly* class. Some of the common attributes have been defined below:

1. AssemblyAlgorithmIdAttribute class – specifies the hash algorithm to use when reading hash files in the assembly manifest
2. AssemblyCompanyAttribute class – specifies the company that produced the assembly
3. AssemblyConfigurationAttribute class – specifies which configuration is used for the assembly (for example, "Debug" or "Release")
4. AssemblyCopyrightAttribute class - specifies copyright information for the assembly
5. AssemblyCultureAttribute class - specifies the culture setting for the assembly
6. AssemblyDefaultAliasAttribute class - specifies a simple name for the assembly
7. AssemblyDelaySignAttribute class - specifies that the assembly will be signed (or strongly named) after it is compiled and marked as a strong assembly
8. AssemblyDescriptionAttribute class - specifies a description for the assembly
9. AssemblyFileVersionAttribute class - specifies the file version for the assembly
10. AssemblyFlagsAttribute class - specifies one or more *AssemblyNameFlags* for the assembly (e.g. EnableJITcompile-Optimizer, EnableJITcompile-Tracking, PublicKey, Retargetable, or None)
11. AssemblyInformationalVersionAttribute class - specifies the version (for informational purposes only) for the assembly
12. AssemblyKeyFileAttribute class - specifies the path of the key file used to sign the assembly
13. AssemblyTitleAttribute class - specifies the title for the assembly
14. AssemblyTrademarkAttribute class - specifies the trademark information for the assembly
15. AssemblyVersionAttribute class - specifies the version for the assembly

Info classes

1. ConstructorInfo class - indicates the attributes of a class constructor, provides access to constructor metadata, and invoke the constructor
2. MethodInfo class – indicates the attributes of a method and provides access to method metadata
3. MemberInfo class - provides information about the attributes of a member and provides access to member metadata. This class is the abstract base class for classes used to obtain information about all members of a class (constructors, events, fields, methods, and properties).
4. PropertyInfo class - indicates the attributes of a property and provides access to property metadata
5. FieldInfo class – indicates the attributes of a field and provides access to field metadata
6. EventInfo class – indicates the attributes of an event and provides access to event metadata
7. LocalVariableInfo class – indicates the attributes of a local variable and provides access to local variable metadata. The class is new to the .NET Framework 2.0.

Binder class and BindingFlags

The *Binder* class determines how to do type conversions and where to locate dynamic code. It selects a member from a list of candidates and performs type conversion from actual argument type to formal argument types.

The *BindingFlags* enumeration specifies flags that control binding. It also indicates the way in which the search for members and types is conducted by reflection. This enumeration has a *FlagsAttribute* attribute which allows a bitwise combination of its member values.

MethodBase class and MethodBody class

The *MethodBase* class provides information about methods and constructors. Also, the *MethodBase* class is the base class of the *MethodInfo* and *ConstructorInfo* classes.

The *MethodBody* class (new to the .NET Framework 2.0) provides access to the metadata (local variables and exception-handling clauses in a method body) and the Microsoft intermediate language (MSIL) for the body of a method. To instantiate a handle on the *MethodBody* object for a given method, you must first obtain a handle to the *MethodInfo* object, then call the *GetMethodBody* method.

Builder classes

When building code at runtime, it is encapsulated as other code would be (an assembly is created, modules within the assembly are created, and types within the modules are created). Builder classes define the types of classes used to build dynamic assemblies, types, methods, etc.

1. AssemblyBuilder class – used to build assemblies at runtime
2. ConstructorBuilder class - used to build constructors at runtime
3. EnumBuilder class - used to build enumerations at runtime
4. EventBuilder class - used to build events at runtime
5. FieldBuilder class - used to build fields at runtime
6. LocalBuilder class - used to build local variables for methods and contructors
7. MethodBuilder class - used to build methods at runtime
8. ModuleBuilder class - used to build modules at runtime
9. ParameterBuilder class - used to build parameters at runtime
10. PropertyBuilder class - used to build properties at runtime
11. TypeBuilder class- used to build types at runtime

7.4 Send electronic mail to a Simple Mail Transfer Protocol (SMTP) server for delivery from a .NET Framework application.

System.Net.Mail namespace

The *System.Net.Mail* namespace provides classes to create and transmit e-mail messages to an SMTP server (Simple Mail Transfer Protocol). The namespace is new to the .NET Framework 2.0.

MailMessage class

The *MailMessage* class (new to the .NET Framework 2.0) represents an e-mail message (sent using the *SmtpClient* class). You can specify sender, recipients, and contents of an e-mail message, attachments, etc. using the properties of the *MailMessage* class shown below:

- From – the sender of the e-mail
- To – the recipient of the e-mail
- CC – carbon copy recipients of the e-mail
- BCC – blind carbon copy recipients of the e-mail
- Attachments – one or more attachments for the e-mail
- Subject – the subject of the e-mail
- Body – the actual message body of the e-mail

MailAddress class and MailAddressCollection class

The *MailAddress* class (new to the .NET Framework 2.0) stores address information for an electronic mail sender or recipient. The *MailAddress* class is used by both the *SmtpClient* and *MailMessage* classes. The mail address is comprised of a *User* name, *Host* name and optionally, a *DisplayName* (which can contain non-ASCII characters if encoded).

The *MailAddressCollection* class (new to the .NET Framework 2.0) stores e-mail addresses that are associated with an e-mail message. This class is used by the *MailMessage.To*, *MailMessage.CC*, and *MailMessage.Bcc* properties.

SmtpClient class, SmtpPermission class, and SmtpPermission Attribute class

The *SMTPClient* class (new to the .NET Framework 2.0) provides the ability for applications to send e-mail using the Simple Mail Transfer Protocol (SMTP). When using the *SMTPClient* class to send an e-mail, the following information must be provided:

- The SMTP host server
- Credentials for authentication (if required by the SMTP server)
- The e-mail address of the sender

- The e-mail address (or addresses) of the recipients
- The message content

The *SmtpPermission* class (new to the .NET Framework 2.0) manages access to Simple Mail Transport Protocol (SMTP) servers.

The *SmtpPermissionAttribute* class (new to the .NET Framework 2.0) controls access to Simple Mail Transport Protocol (SMTP) servers.

Attachment class, AttachmentBase class, and AttachmentCollection class

The *MailAddress* class (new to the .NET Framework 2.0) represents an file attachment included in an e-mail. When adding or more attachments to an e-mail, the file attachments are added to the *MailMessage.Attachments* collection and the content of the attachment can be a *String*, *Stream*, or file name.

The *AttachmentBase* class (new to the .NET Framework 2.0) is a base class that represents an email attachment. The *Attachment*, *Alternateview*, and *LinkedResource* classes all derive from this class.

The *AttachmentCollection* class (new to the .NET Framework 2.0) contains one or more attachments to be sent as part of an e-mail message.

SmtpException class and SmtpFailedReceipientException class

The *SmtpException* class (new to the .NET Framework 2.0) is the exception thrown when the *SmtpClient* object is unable to complete a *Send* or *SendAsync* operation. You can use the *SmtpException.StatusCode* class (new to the .NET Framework 2.0) returned by SMTP server when an e-mail message is transmitted to determine the details of the error.

The *SmtpFailedRecipientException* class (new to the .NET Framework 2.0) is the exception thrown when the *SmtpClient* is not able to complete the *Send* or *SendAsync* operation.

SendCompletedEventHandler delegate

The *SendCompletedEventHandler* delegate (new to the .NET Framework 2.0) handles events that occur when the *SmtpClient* class finishes sending an e-mail message using the *SendAsync* method (asynchronously) and the *SendCompleted* event is triggered.

LinkedResource class and LinkedResourceCollection class

The *LinkedResource* class (new to the .NET Framework 2.0) represents an embedded external resource in an email attachment (e.g. an image in an HTML attachment).

The *LinkedResourceCollection* class (new to the .NET Framework 2.0) contains objects that store linked resources sent as part of an e-mail message. Each instantiation of the *LinkedResourceCollection* class is returned by the *AlternateView.LinkedResources* property

AlternateView class and AlternateViewCollection class

The *AlternateView* class (new to the .NET Framework 2.0) contains an object that represents the format to view an email message. The *AlternateView* class is used to specify copies of the e-mail message in either an HTML format or plain text format. This should always be used when sending e-mails to users whose e-mail clients may not support HTML formatted e-mails.

The *AlternateViewCollection* class (new to the .NET Framework 2.0) represents a collection of *AlternateView* objects. The *AlternateView* class is used to indicate an e-mail message in different formats (HTML and plain text).

Implementing globalization, drawing, and text manipulation functionality in a .NET Framework application

8.1 Format data based on culture information.

System.Globalization namespace

The *System.Globalization* namespace represents the classes that define culture information allowing for the development of applications for multiple, geographical disperse (international) regions. The *System.Globalization* namespace provides capabilities to develop applications that specify settings for language, country/region, calendars, format patterns for dates, currency, numbers, and sort orders for strings.

Access culture and region information in a .NET Framework application.

CultureInfo class

The *CultureInfo* class provides information about the culture, or locale, context in which the application is running. Culture specific information, such as settings for language, sub-language, country/region, calendars, format patterns for dates, currency, numbers, casing, and string comparisons are available.

Note: The *String* class uses this class in order to obtain information regarding the default culture

A unique name is specified for each culture. This unique name is a combination of an ISO 639 two-letter lowercase culture code associated with a language and an ISO 3166 two-letter uppercase subculture code associated with a country or region. For example, "en-US" represents U.S. English. A neutral culture is specified by only the two-digit lowercase language code. For example, "en" will specify the neutral culture for English. For a complete list of the predefined culture names and identifies, refer to the Microsoft website: http://msdn2.microsoft.com/en-us/library/system.globalization.cultureinfo.aspx.
This class is typically used to do the following:

- Control how string comparisons are performed
- Control how number comparisons and formats are performed
- Control how date comparisons are performed
- Control how resources are retrieved and used

Cultures are typically grouped into three categories:

1. Invariant Culture – this is culture-insensitive and used as a default culture when consistency is desired. An Invariant culture is indicated by using an empty string ("") or the culture identifier "0x007F".
2. Neutral Culture – associated with a language but has no relationship to countries or regions (such as English – "en", French – "fr", and Spanish – "sp").
3. Specific Culture – the most precise category in which a neutral culture and specific culture is indicated (for example "en-US"). When developing applications, specific cultures should be indicated whenever possible.

CultureTypes enumeration

The *CultureTypes* enumeration represents type of culture lists (defined by the *CultureInfo.CultureTypes* property) that can be retrieved using the *CultureInfo.GetCultures* method. This enumeration is marked with the *FlagsAttribute* so is allows the specification of multiple values.

The *CultureInfo.CultureTypes* class (new to the .NET Framework 2.0) return the culture types for the current *CultureInfo* object.

RegionInfo class

The *RegionInfo* class contains detailed information about the country or region. The *RegionInfo* name is one of the two-letter codes defined in ISO 3166 for country/region.

Format date and time values based on the culture.

DateTimeFormatInfo class

The *DateTimeFormatInfo* class indicates how *DateTime* values are formatted and displayed based on the culture (such as date patterns, time patterns, and AM/PM designators). In order to create a *DateTimeFormatInfo* object for a specific culture, create a *CultureInfo* object for the culture and then retrieve the *CultureInfo.DateTimeFormat* property. For a detailed list of *DateTime* format specifiers and their associated *DateTimeFormatInfo* properties, refer to the Microsoft web address: http://msdn2.microsoft.com/en-us/library/system.globalization.datetimeformatinfo.aspx.

Format number values based on the culture.

NumberFormatInfo class

The *NumberFormatInfo* class indicates how numeric values (currency, decimal separators, and other numeric symbols) are formatted and displayed based on the culture settings.). In order to create a *NumberFormatInfo* object for a specific culture, create a *CultureInfo* object for the culture and then retrieve the *CultureInfo.NumberFormat* property. For a detailed list of *NumberFormatInfo* format characters for each specified pattern and their associated *NumberFormatInfo* properties, refer to the Microsoft web address: http://msdn2.microsoft.com/en-us/library/system.globalization.numberformatinfo.aspx.

NumberStyles enumeration

The *NumberStyles* enumeration indicates the styles permitted in numeric string arguments passed to the *Parse* method of the numeric base type classes. For example, currency symbols, thousands separators, decimal point indicators, and leading signs. This enumeration is marked with the *FlagsAttribute* so is allows for bitwise combinations of member values. For a detail list of styles, refer to the Microsoft web address: http://msdn2.microsoft.com/en-us/library/system.globalization.numberstyles.aspx.

Perform culture-sensitive string comparison.

CompareInfo class
The *CompareInfo* class implements a set of methods for culturally sensitive string comparisons. A new instance of the *CompareInfo* class can be created or use the *CompareInfo* property of the *CultureInfo* class to get a handle on the *CompareInfo* object. The *GetCompareInfo* method allows for late-bound access (rather than using a public contstructor).

Note: The *String.Compare* uses the information in *CultureInfo.CompareInfo* to compare strings

CompareOptions enumeration
The *CompareOptions* enumeration controls how comparisons are performed on the *CompareInfo* class. For example, case sensitivity or whether types of characters should be ignored. This enumeration is marked with the *FlagsAttribute* so is allows for bitwise combinations of member values.

Build a custom culture class based on existing culture and region classes.

CultureAndRegionInfoBuilder class
The *CultureAndRegionInfoBuilder* class (new to the .NET Framework 2.0) allows programmers to create and use customized cultures. Developers can define a custom culture that is new or overrides an existing culture (based on an existing culture and region). This custom culture can then be installed on a system and subsequently used by any application running on that system (as long as the user has administrative rights to that computer).

CultureAndRegionModifier enumeration
The ***CultureAndRegionModifier*** enumeration (new to the .NET Framework 2.0) indicate the constants that define the ***CultureAndRegionInfoBuilder*** object. The members made available by this enumeration are:

- Neutral – a neutral custom culture
- None - a specific, supplemental custom culture
- Replacement – a custom culture that replaces an existing .NET Framework culture or Windows Locale

This enumeration is marked with the ***FlagsAttribute*** so is allows for bitwise combinations of member values.

8.2 Enhance the user interface of a .NET Framework application by using the System.Drawing namespace.

The *System.Drawing* namespace provides programmatic access to CDI+ basic graphics functionality so that you can draw to the display device, which allows you to create new images or modify existing images. More advanced capabilities are available in the following derived namespaces:

- System.Drawing.Drawing2D
- System.Drawing.Imaging
- System.Drawing.Text

Some function made available by the *System.Drawing* namespace are:

- Add circles, lines, and other shapes dynamically
- Create charts
- Edit and resize pictures
- Change compression ratios of pictures saved to disk
- Crop or zoom images
- Add copyright logos to text or pictures

While there are over a dozen classes made available by the *System.Drawing*, namespace, some of the more important and common classes are discussed below.

Enhance the user interface of a .NET Framework application by using brushes, pens, colors, and fonts.

Brush class
The *Brush* class defines objects to fill the interior of graphical shapes (rectangles, ellipsis, pies, polygons, and paths). Because this is an abstract class, *Brush* objects cannot be instantiated and must be derived (from *SolidBrush*, *TextureBrush*, and *LinearGradientBrush*).

Brushes class
The *Brushes* class provides the brushes for all standard colors. It is a static class that provides read-only properties returning a *Brush* object of the color represented by the property name.

SystemBrushes class
The *SystemBrushes* class represents a *SolidBrush* where each property is a color of a Windows display element. This class cannot be inherited.

TextureBrush class
The *TextureBrush* class represents a *Brush* object that uses an image to fill the interior of a shape.

Pen class
The *Pen* class defines an object used to draw lines, curves, and arrows with a specified width and style. The line of the Pen object can be filled with solid colors, textures, and fill styles (depending on the brush or textures used to fill the object). You can also use the *DashStyle* to draw different types of dashed lined. This class cannot be inherited.

Pens class
The *Pen* class provides access to *Pen* objects for all of the standard colors. The Pen objects are immutable (so their properties cannot modified). This class cannot be inherited.

SystemPens class
The *SystemPens* class provides access to the *Pen* class where each property of the *System.Pens* object is the color of the Windows display element having the width of 1 pixel.

SolidBrush class
The *SolidBrush* class represents a Brush of a single color used to fill graphic shapes (such as rectangles, ellipses, pies, polygons, etc.). This class cannot be inherited.

Color structure
The *Color* structure represents a color and is used to specify a control's color. You can specify the color of a control by using the predefined properties available by the *System.Drawing.Color* class or you can specify a custom color using the *Color.Argb* method.

ColorConverter class
The *ColorConverter* class converts colors from one data type to another. This class expects an unqualified color name (for example, "black") rather than a qualified name (for example, "System.Drawing.Color.Black") or an exception is thrown during the conversion call. This class is access through the *TypeDescriptor* class.

ColorTranslator class
The *ColorTranslater* class is used to translate colors from one GDI+ color structure to another GDI+ color structure. This class cannot be inherited.

SystemColors class
The *SystemColors* class provides access to the *Color* structure where each property of the *SystemColors* class is the color of the Windows display element.

StringFormat class
The *StringFormat* class provides access to the text layout information (alignment, orientation, tab stops, etc) and display features (ellipses insertions and national digits substitution) and *OpenType* features. This class cannot be inherited.

Font class
The *Font* class defines a specific font format for text such as font face, font size, and style attributes. This class cannot be inherited.

Note: Windows form display true type fonts and only support OpenType fonts on a limited basis. If you attempt to use a font that is not supported (or installed on the local machine), the system will default to the Microsoft Sans Serif font.

FontConverter class
The *FontConverter* class converts *Font* objects from one type to another. This class is accessed via the *TypeDescriptor* object.

FontFamily class
The *FontFamily* class defines a group of typefaces having similar font designs as well as font typefaces that have certain variations in styles. This class cannot be inherited.

SystemFonts class
The *SystemFonts* (new to the .NET Framework 2.0) specified the fonts used to display text within Windows display elements. Each property of the *SystemFonts* class returns a Font object which displays text in a particular Windows display element. The font settings indicated are also represented in the settings located in the systems Control Panel.

Enhance the user interface of a .NET Framework application by using graphics, images, bitmaps, and icons.

Graphics class
The *Graphics* class provides methods for drawing to the display device by encapsulating the GDI+ drawing surface. This class cannot be inherited. A handle to eh *Graphics* object can be obtained by:

1. Calling the *Control.CreateGraphics* method on an object that inherits from *System.Windows.Forms.Control*
2. A control's *Control.Paint* event
3. Accessing the Graphics property of the *System.Windows.Forms.PaintEventArgs* class

BufferedGraphics class
The *BufferredGraphics* class (new to the .NET Framework 2.0) provides a graphics buffer for double buffering. This provides the programmer a wrapper for a graphics buffer, methods to write to the graphics buffer, and render the contents of the buffer to an output device.

Note: Graphical double buffering helps to reduce (or eliminate) image flickering caused when redrawing a display surface. Updated images are first drawn to the buffer in memory and its contents are quickly written to the displayed surface. This additional overwriting of the images helps to reduce the visual flickering noticed by the application users.

The The *BufferredGraphics* object is created using the *BufferredGraphicsContext* for the application using the *Allocate* method.

BufferedGraphicsManager class
The *BufferedGraphicsManager* class (new to the .NET Framework 2.0) provides access to the main buffered graphics context object in the application domain. The *BufferredGraphics* class provides a graphics buffer for double buffering to help to reduce (or eliminate) image flickering. The *Current* property of the *BufferedGraphicsManager* class returns the main *BufferredGraphicsContext* object for the current application domain which can be used to create *BufferredGraphics* objects to draw buffered graphics.

Image class
The *Image* class is an abstract base class that provides functionality for the *Bitmap* and *Metafile* descendent classes. Using this class, you can create, load, modify, and save images of various image types (bitmaps, jpegs, tiff files, etc.). You can perform sophisticated image functions such as drawing objects, images, and charts and then saving them as image files. This class can also be used to write copyright information or watermarks to pictures and resize images to consume less space.

New instances of the *Image* class can be created using the *Image.FromFile* and *Image.FromStream* methods or by inheriting the image from the *System.Drawing.Bitmap* or *System.Drawing.Imaging.Metafile* classes.

ImageConverter class
The *ImageConverter* class converts an *Image* object from one data type to another. This class is accessed using the *TypeDescriptor* object.

ImageAnimator class
The *ImageAnimator* class animates an image that has time-based frames.

Bitmap class
The *Bitmap* class is used when working with new or existing images (bitmaps represented by pixel data and attributes). This class encapsulates GDI+ bitmaps (pixel data for graphics images and their attributes). A new *Bitmap* object can be created from an existing image, existing file, stream object, or as a blank bitmap of a specified height and width.

Icon class
The *Icon* class represents a Windows icon (whose size is determined by the system).

IconConverter class
The *IconConverter* class converts an icon from one data type to another. This class is accessed using the *TypeDescriptor* object.

SystemIcons class
The *SystemIcon* class represents the *Icon* object whose properties are Windows system-wide icons. This class cannot be inherited. The .NET Framwork provides 40 pixel by 40 pixel system icons as properties of the *SystemIcon* class.

Enhance the user interface of a .NET Framework application by using shapes and sizes.

Point Structure
The *Point* structure represents an ordered pair of x and y coordinates which, together, define a point on a two dimensional plane.

PointConverter class
The *PointConverter* class provides the ability to convert a *Point* object from one data type to another. This class is accessed using the *TypeDescriptor* object.

Rectangle Structure
The *Rectangle* structure indicates a set of four integers that represent the location and size of a rectangle (which is defined by its height, width, and location as the upper-left corner).

RectangleConverter class
The *RectangleConverter* class converts rectangles from one data type to another. This class is accessed using the *TypeDescriptor* object.

Size Structure
The *Size* structure stores an ordered pair of integers that can be used to represent the width and height of a rectangle.

SizeConverter class
The *SizeConverter* class is used to convert from one data type to another. This class is accessed using the *TypeDescriptor* object.

Region class
The *Region* class represents the interior of a graphics object which is composed of rectangles and paths. This class cannot be inherited.

8.3 Enhance the text handling capabilities of a .NET Framework application (refer System.Text namespace), and search, modify, and control text in a .NET Framework application by using regular expressions.

System.Text.RegularExpressions

A regular expression is a set of characters that can be compared to another string in order to determine if the string meets specific format requirements or regular expressions can be used to extract/replace portions of text within a string. Regular expressions can be matched against strings consisting of integers, strings containing lowercase letters, or string matching hexadecimal output. Blocks of text can be extracted from strings or updated to modify the format of text or remove specified characters. The *System.Text.RegularExpressions* namespace contains classes which provide access to the .NET Framework regular expression engine. Some of these classes are described in this chapter.

StringBuilder class

The *StringBuilder* class represents a string object whose value is a mutable sequence of characters. Most of the methods of this class that modify the value of the object, return a reference to the same instance of the class (as opposed to creating a new instance of the class). This class cannot be inherited.

Note: A mutable class is a class where the value can be modified once it has been created by appending, removing, replacing, or inserting characters.

The *StringBuilder* class is unlike the *String* object which is immutable and, therefore, creates a new string object in memory whenever a method of the *String* object is called. You can think of immutable objects as read-only objects while mutable objects can be updated.

The default capacity (maximum number of characters the instance can store at any given time) of the *StringBuilder* object is 16. However, the *StringBuilder* class can allocate more memory as needed. The largest value for capacity is represented by Int32 (or 2,147,483,647). In addition, the *Capacity* and *Length* properties can be set (or read) programmatically.

The contents of the *StringBuilder* class can be modified using the following methods:

- Append - add text or a string representation of an object to the end of a string represented by the current *StringBuilder*
- AppendFormat - adds text to the end of the *StringBuilder* implementing the *IFormattable* interface
- Insert - adds a string or object to a specified position in the current *StringBuilder*
- Remove - remove a specified number of characters from the current *StringBuilder*, beginning at a specified zero-based index
- Replace – replace characters within the *StringBuilder* object with another specified character

Regex class

The *RegEx* class represents an immutable regular expression. However, the *RegEx* class does have several static methods which allow you to use a regular expression without explicitly creating a new *Regex* object (a result of being an immutable, read-only object).

Match class and MatchCollection class
The *Match* class represents the results from a single regular expression match (which is immutable). A single *Match* can involve multiple matching groups. When this occurs, the *Groups* property returns a *GroupCollection* (described below).

The *MatchCollection* class represents a set of successful matches (which are immutable). These matches are found by iteratively applying a regular expression pattern to a input string.

Group class and GroupCollection class
The *Group* class represents the results from a single capturing group (zero, one, or more strings in a single match) and supplies a collection of *Capture* objects. The *Group* class inherits from the *Capture* class and the last substring captured can be accessed directly.

The *GroupCollection* class represents a collection of *Groups* by returning the set of captured *Groups* in a single match. Instanced are immutable and are only returned in the collection of *Groups* objects.

Encode text by using Encoding classes
The .NET Framework provides the ability for programmers to manually encode and decode characters. Encoding is defined as the process of transforming a set of Unicode characters into a sequence of bytes and decoding is the process of transforming a sequence of encoded bytes into a set of Unicode characters.

Encoding class
The *Encoding* class uses the Unicode standard to assign a number to each character. The Unicode Standard Version 2.3 uses the following UTFs (Unicode Transformation Format which is used for encoding):

- Unicode UTF-8 encoding - encodes Unicode characters using the UTF-8 encoding (represents each code point as a sequence of one to four bytes). This encoding supports all Unicode character values.
- Unicode UTF-16 encoding - encodes Unicode characters using the UTF-16 encoding (represents each code point as a sequence of one to two 16-bit integers).
- Unicode UTF-32 encoding - encodes Unicode characters using the UTF-32 encoding (represents each code point as a 32-bit integer).
- ASCIIEncoding - encodes Unicode characters as single 7-bit ASCII characters.
- UTF7Encoding - encodes Unicode characters using the UTF-7 encoding.

Note: If you need to encode/decode non Unicode characters (such as binary data), use an alternate protocol such as uuencode.

EncodingInfo class
The *EncodingInfo* class (new to the .NET Framework 2.0) provides basic information about encoding. Used by the *Encoding* class, the *EncodingInfo* class provides minimal information about the encoding such as the code page identifier for the encoding , the display name, and the name registered with the Internet Assigned Numbers Authority (IANA) for the encoding. More detailed encoding information can be accessed using the *GetEncoding* method of *Encoding* class.

ASCIIEncoding class
The *ASCIIEncoding* class represents the ASCII character encoding of Unicode characters. *ASCIIEncoding* only supports Unicode character values between U+0000 and U+007F and should therefore not be used for global applications.

UnicodeEncoding class
The *UnicodeEncoding* class represents UTF-16 encoding of Unicode characters where each code point as a sequence of one to two 16-bit integers.

UTF8Encoding class
The *UTF8Encoding* class represents UTF-8 encoding of Unicode characters where each code point as a sequence of one to four bytes.

Encoding Fallback classes
The *EncoderFallback* class (new to the .NET Framework 2.0) implements a failure handling framework for an input character that cannot be converted to an encoded output byte sequence. IN essence, it provides a "fallback" mechanism. You can use predefined .NET Framework encoder fallbacks or, alternatively, create custom encoder fallbacks (derived from the *EncoderFallback* and *EncoderFallbackBuffer* classes).
There are two difference fallback strategies available.

1. Use the *EncoderReplacementFallback* class which substitutes a string you provide for any input character that cannot be converted; the string is encoded instead of the invalid character
2. Use the *EncoderExceptionFallback* class which will throw an *EncoderFallbackException* when an invalid character is encountered

Decode text by using Decoding classes.

Decoder class
The *Decoder* class converts a sequence of encoded bytes into a set of characters.

Decoder Fallback class
The *DecoderFallback* class (new to the .NET Framework 2.0) implements a failure handling framework for an encoded input byte sequence that cannot be converted to an output character. You can use predefined .NET Framework decoder fallbacks or, alternatively, create custom decoder fallbacks (derived from the *DecoderFallback* and *DecoderFallbackBuffer* classes).

Capture class and CaptureCollection class
The *Capture* class contains the results as a single substring from a successful, sub expression capture. The instances are returned through a collection, accessible via the *Captures* class. This class is immutable.
The *CaptureCollection* class returns a set of *Capture* substrings collected by one capturing group. The instances are returned through the *Captures* collection. This class is immutable.