МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

Нижегородский государственный университет им. Н.И. Лобачевского

С.В. Сидоров

.Net технологии

Учебно-методическое пособие

Рекомендовано методической комиссией факультета ВМК для иностранных студентов, обучающихся в ННГУ по направлению подготовки 010300 «Фундаментальная информатика и информационные технологии» (бакалавриат)

1-е издание

Нижний Новгород

2011

УДК 53.088

ББК В3В6

Ф-15

Ф-15 Сидоров С.В. .NET ТЕХНОЛОГИИ: Учебное пособие. – Нижний Новгород: Нижегородский госуниверситет, 2011. – 36 с.

Рецензенты: профессор **В.А. Перевощиков**,

кандидат физ.-мат. наук **В.А. Гришагин**

В настоящем пособии изложены учебно-методические материалы по курсу “.Net технологии” для иностранных студентов, обучающихся в ННГУ по направлению подготовки 010300 «Фундаментальная информатика и информационные технологии» (бакалавриат).

Учебно-методическое пособие предназначено для студентов факультета иностранных студентов обучающихся по направлению подготовки 010300 «Фундаментальная информатика и информационные технологии» и может быть использовано школьниками старших классов, занимающихся научной работой в рамках НОУ.

УДК 53.088

ББК В3В6

Ministry of Education and Science of the Russian Federation

State educational institution of higher education
«Lobachevsky State University of Nizhni Novgorod»

S.V.Sidorov

.Net technologies

Study book

Recommended by the Methodical Commission of the Faculty of Computer Science for international students, studying at the B.Sc. programme "010300"

"Fundamental Informatics and Information Technologies"

Nizhny Novgorod

2011

Contents

Paragraph 1. Course objectives

This course is designed for students who need to know how to develop applications using the Microsoft .NET Framework 2.0. We assume that before you begin using this course you have a working knowledge of Microsoft Windows and Microsoft C#. Participating this course, you'll learn how to do the following:

- Develop applications that use system types and collections.
- Implement service processes, threading, and application domains to enable application isolation and multithreading.
- Create and deploy manageable applications.
- Create classes that can be serialized to enable them to be easily stored and transferred.
- Create hardened applications that are resistant to attacks and restrict access based on user and group roles.
- Use interoperability and reflection to leverage legacy code and communicate with other applications.
- Write applications that send e-mail messages.
- Draw charts and create images, and either display them as part of your application or save them to files.

Paragraph 2. Lessons notes

Lesson 1: Using Value Types

The simplest types in the .NET Framework, primarily numeric and Boolean types, are value types. Value types are variables that contain their data directly instead of containing a reference to the data stored elsewhere in memory. Instances of value types are stored in an area of memory called the *stack*, where the runtime can create, read, update, and remove them quickly with minimal overhead.

There are three general value types:

- Built-in types
- User-defined types
- Enumerations

Each of these types is derived from the *System.Value* base type. The following sections show how to use these different types.

After this lesson, you will be able to:

- Choose the most efficient built-in value type
- Declare value types
- Create your own types
- Use enumerations

Estimated lesson time: 110 minutes

Lesson 2: Using Common Reference Types

Most types in the .NET Framework are reference types. Reference types provide a great deal of flexibility, and they offer excellent performance when passing them to methods. The following sections introduce reference types by discussing common built-in classes. Lesson 4, "Converting Between Types," covers creating classes, interfaces, and delegates.

After this lesson, you will be able to:

- Explain the difference between value types and reference types.
- Describe how value types and reference types differ when assigning values.
- List the built-in reference types.
- Describe when you should use the *StringBuilder* type.
- Create and sort arrays.
- Open, read, write, and close files.
- Detect when exceptions occur and respond to the exception.

Estimated lesson time: 90 minutes

Lesson 3: Constructing Classes

In object-oriented languages, the bulk of the work should be performed within objects. All but the simplest applications require constructing one or more custom classes, each with multiple properties and methods used to perform tasks related to that object. This lesson discusses how to create custom classes.

After this lesson, you will be able to:

- Describe and use inheritance.
- Describe and use interfaces.
- Describe and use partial classes.
- Create a generic type, and use the built-in generic types.
- Respond to and raise events.
- Add attributes to describe assemblies and methods.
- Move a type from one class library to another using type forwarding.

Estimated lesson time: 110 minutes

Lesson 4: Converting Between Types

Often, you need to convert between two different types. For example, you might need to determine whether an *Integer* is greater or less than a *Double*. You might need to pass a *Double* to a method that requires an *Integer* as a parameter. Or you might need to display a number as text. This lesson describes how to convert between types in both Visual Basic and C#. Type conversion is one of the few areas where Visual Basic and C# differ considerably.

After this lesson, you will be able to:

- Convert between types.
- Explain boxing and why it should be avoided.
- Implement conversion operators.

Estimated lesson time: 120 minutes

Lesson 5. Navigating the File System

In the everyday work of developers, one of the most common tasks is to work with the file system. This task includes navigating and gathering information about drives, folders, and files as well as waiting for changes to happen in the file system.

After this lesson, you will be able to:

- Write code that uses the *File* and *FileInfo* classes.
- Write code that uses the *Directory* and *DirectoryInfo* classes.
- Use the *DriveInfo* and *DriveType* classes.
- Enumerate files, directories, and drives using the *FileSystemInfo* derived classes.
- Use the *Path* class to manipulate file system paths.
- Watch for changes in the file system using the *FileSystemWatcher* class.

Estimated lesson time: 120 minutes

Lesson 6: Reading and Writing Files

Reading and writing files are two of the most common tasks in the world of development. As a .NET developer, you need to know how to read and write files. The .NET
Framework makes it easy to perform these tasks.

After this lesson, you will be able to:

- Open a file and read its contents.
- Create an in-memory stream.
- Write and close a file.

Estimated lesson time: 120 minutes

Lesson 7: Compressing Streams

Now that you know the basics of how to work with streams, you're ready to learn about a new type of stream that will be important in certain types of projects. Often in real-world projects, you will find it useful to save space or bandwidth by compressing data. The .NET Framework supports two new stream classes that can compress data.

After this lesson, you will be able to:

- Compress streams with the *GZipStream* and *DeflateStream* classes.
- Decompress streams with the *GZipStream* and *DeflateStream* classes.

Estimated lesson time: 110 minutes

Lesson 8: Working with Isolated Storage

As we are becoming more and more aware of, giving programs unfettered access to a computer is not a great idea. The emergence of spyware, malware, and viruses tells us that working in the sandbox of limited security is a better world for most users. Unfortunately, many programs still need to save some sort of state data about themselves.

The way to do this can be as innocuous as storing data in a cache. To bridge the needs of applications to save data and the desire of administrators and users to use more limited security settings, the .NET Framework supports the concept of *isolated storage*.

After this lesson, you will be able to:

- Access isolated storage for storage of program data by using the *IsolatedStorageFile* class.
- Create files and folders in isolated storage by using the *IsolatedStorageFileStream* class.
- Access different stores within isolated storage on a per-user and per-machine basis using the *IsolatedStorageFile* class.

Estimated lesson time: 115 minutes

Lesson 9: Forming Regular Expressions

Developers frequently need to process text. For example, you might need to process input from a user to remove or replace special characters. Or you might need to process text that has been output from a legacy application to integrate your application with an existing system. For decades, UNIX and Perl developers have used a complex but efficient technique for processing text: regular expressions. A regular expression is a set of characters that can be compared to a string to determine whether the string meets specified format requirements. You can also use regular expressions to extract portions of the text or to replace text. To make decisions based on text, you can create regular expressions that match strings consisting entirely of integers, strings that contain only lowercase letters, or strings that match hexadecimal input. You can also extract key portions of a block of text, which you could use to extract the state from a user's address or image links from an HTML page. Finally, you can update text using regular expressions to change the format of text or remove invalid characters.

After this lesson, you will be able to:

- Use regular expressions to determine whether a string matches a specific pattern.
- Use regular expressions to extract data from a text file.
- Use regular expressions to reformat text data.

Estimated lesson time: 145 minutes

Lesson 10: Encoding and Decoding

Every string and text file is encoded using one of many different encoding standards. Most of the time, the .NET Framework handles the encoding for you automatically. However, there are times when you might need to manually control encoding and decoding, such as when:

- Interoperating with legacy or UNIX systems
- Reading or writing text files in other languages
- Creating HTML pages
- Manually generating e-mail messages

This lesson describes common encoding techniques and shows you how to use them in .NET Framework applications.

After this lesson, you will be able to:

- Describe the importance of encoding, and list common encoding standards.
- Use the *Encoding* class to specify encoding formats, and convert between encoding standards.
- Programmatically determine which code pages the .NET Framework supports.
- Create files using a specific encoding format.
- Read files using unusual encoding formats.

Estimated lesson time: 130 minutes

Lesson 11: Collecting Data Items

Computers are naturally good at dealing with large amounts of data. In your daily work as a developer, you will find it necessary to store data in an orderly way. The .NET Framework supports dealing with data in this way by providing a wide range of collections to store your data in. For every collection job, the .NET Framework supplies
a solution.

After this lesson, you will be able to:

- Create a collection.
- Add and remove items from a collection.
- Iterate over items in a collection

Estimated lesson time: 115 minutes

Lesson 12: Working with Sequential Lists

Not all collections are created equal. At times, it makes more sense to deal with a list of objects as a sequential list of items rather than accessing them individually.

After this lesson, you will be able to:

- Create and use a first-in, first-out (FIFO) collection
- Create and use a last-in, first-out (LIFO) collection.

Estimated lesson time: 110 minutes

Lesson 13: Working with Dictionaries

At the other end of the spectrum from sequential lists are dictionaries. Dictionaries are collections that are meant to store lists of key/value pairs to allow lookup of values based on a key.

After this lesson, you will be able to:

- Use a *Hashtable* to create a simple list of unique items.
- Use a *SortedList* to sort a list of objects.
- Work with *DictionaryEntry* objects to store name/value pairs.
- Enumerate dictionaries and know how to use *DictionaryEntries*.
- Understand the *IEqualityComparison* interface to provide uniqueness to Hashtables.
- Use the *HybridDictionary* to store name/value pairs in a very efficient way.
- Use the *OrderedDictionary* to store name/value pairs in a way that preserves the order of adding them to the collection.

Estimated lesson time: 120 minutes

Lesson 14: Using Specialized Collections

The previous three lessons introduced a series of collections that can be used to store any object in .NET. Although these are valuable tools, using them can often lead to you having to cast objects when you retrieve them from the collections. The .NET Framework supports a new namespace called *System.Collections.Specialized* that includes collections that are meant to work with specific types of data.

After this lesson, you will be able to:

- Use the *BitArray* and *BitVector32* classes to deal with sets of *Boolean* values.
- Use the *StringCollection* and *StringDictionary* classes to store collections of strings.
- Use the *NameValueCollection* to store name/value pairs in a type-safe way.

Estimated lesson time: 150 minutes

Lesson 15: Generic Collections

Prior to Lesson 14, only collections that worked with objects were discussed. If you wanted to retrieve a specific type of object, you needed to cast that object to its real type. In Lesson 14, you saw some common specialized collections for working with well-known types such as strings. But adding a couple of specialized collections does not solve most problems with type safety and collections. That is where generic collections come in.

After this lesson, you will be able to:

- Create and work with type-safe lists
- Create and work with type-safe queues
- Create and work with type-safe stacks
- Create and work with type-safe dictionaries
- Create and work with type-safe linked list collections

Estimated lesson time: 120 minutes

Lesson 16: Drawing Graphics

You can use the .NET Framework to enhance the user interface by drawing lines, circles, and other shapes. With just a couple of lines of code, you can display these graphics on a form or other Windows Forms control.

After this lesson, you will be able to:

- Describe the members of the *System.Drawing* namespace.
- Control the location, size, and color of controls.
- Draw lines, empty shapes, and solid shapes.
- Customize pens and brushes to enhance graphics.

Estimated lesson time: 120 minutes

Lesson 17: Working with Images

Often developers need to display, create, or modify images. The .NET Framework provides tools to work with a variety of image formats, enabling you to perform many common image-editing tasks.

After this lesson, you will be able to:

- Describe the purpose of the *Image* and *Bitmap* classes.
- Display pictures in forms or *PictureBox* objects.
- Create a new picture, add lines and shapes to the picture, and save it as a file.

Estimated lesson time: 100 minutes

Lesson 18: Formatting Text

Developers often add text to images to label objects or create reports. This lesson describes how to add formatted text to images.

After this lesson, you will be able to:

- Describe the process of creating the objects required to add text to images.
- Create *Font* objects to meet your requirements for type, size, and style.
- Use *Graphics.DrawString* to annotate images with text.
- Control the formatting of text.

Estimated lesson time: 100 minutes

Lesson 19: Creating Threads

Threads are the basis of high-performance applications. In the .NET Framework, the *System.Threading* namespace contains the types that are used to create and manage multiple threads in an application.

After this lesson, you will be able to:

- Create threads to do work concurrently.
- Start and join threads.
- Abort threads.
- Use critical regions.

Estimated lesson time: 120 minutes

Lesson 20: Sharing Data

The most challenging part of working with threads is sharing data between multiple threads. Once you start to work with multiple threads in an application, you become responsible for the protection of any shared data that can be accessed from multiple threads.

After this lesson, you will be able to:

- Use the *Interlock* class to perform atomic operations.
- Use the C# *lock* or the Visual Basic *SyncLock* syntax to lock data.
- Use the *Monitor* class to lock data.
- Use a *ReaderWriterLock* to lock data.

■ Use a *Mutex* to synchronize threads.
■ Use a *Semaphore* to throttle threads.
■ Use an *Event* to signal threads.

Estimated lesson time: 90 minutes

Lesson 21: Creating a Mail Message

Creating an e-mail message can be simple or complex. At its simplest, an e-mail message has a sender, recipient, subject, and body. These simple messages can be created with a single line of code using the .NET Framework. At their most complex, e-mail messages can have custom encoding types, multiple views for plain text and HTML, attachments, and images embedded within HTML.

After this lesson, you will be able to:

■ Describe the process of creating and sending an e-mail.
■ Create a *MailMessage* object.
■ Attach one or more files to an e-mail message.
■ Create HTML e-mails with or without pictures.
■ Catch and respond to different exceptions that might be thrown while creating a
message.

Estimated lesson time: 120 minutes

Lesson 22: Sending Mail

Often, sending an e-mail message is simple and requires only two lines of code. However, as with any network communication, it can become much more complex. If the server is unresponsive, you need to allow users to decide whether to wait for or cancel the message transfer. Some servers require valid user credentials, so you might need to provide a username and password. When possible, you should enable the Secure Sockets Layer (SSL) to encrypt the message in transit and reduce your security risks.

After this lesson, you will be able to:

■ Configure an SMTP server, and send an e-mail message.
■ Provide a username and password when required to authenticate to an SMTP server.
■ Enable SSL to encrypt SMTP communications.
■ Send a message asynchronously to allow your application to respond to the user or perform other tasks while an e-mail is being sent.
■ Catch and respond to different exceptions that might be thrown while sending a message.

Estimated lesson time: 110 minutes

Paragraph 3. Exercises

Exercise 1: Create a Structure

In this exercise, you will create a simple structure with several public members.

1. Using Visual Studio, create a new console application project. Name the project **CreateStruct**.

2. Create a new structure named *Person*, as the following code demonstrates:

```
' VB
Structure Person
End Structure
// C#
struct Person
{
}
```

3. Within the *Person* structure, define three public members:

- *firstName* (a *String*)
- *lastName* (a *String*)
- *age* (an *Integer*)

The following code demonstrates this:

```
' VB
Public firstName As String
Public lastName As String
Public age As Integer
// C#
public string firstName;
public string lastName;
public int age;
```

4. Create a constructor that defines all three member variables, as the following code demonstrates:

```
' VB
Public Sub New(ByVal _firstName As String, ByVal _lastName As
String, ByVal _age As
Integer)
firstName = _firstName
lastName = _lastName
age = _age
End Sub
// C#
public Person(string _firstName, string _lastName, int _age)
{
     firstName = _firstName;
     lastName = _lastName;
     age = _age;
}
```

5. Override the *ToString* method to display the person's first name, last name, and age. The following code demonstrates this:

```
' VB
Public Overloads Overrides Function ToString() As String
```

```
Return firstName + " " + lastName + ", age " + age.ToString
End Function
// C#
public override string ToString()
{
     return firstName + " " + lastName + ", age " + age;
}
```

6. Within the *Main* method of the console application, write code to create an instance of the structure and pass the instance to the *Console.WriteLine* method, as the following code demonstrates:

```
' VB
Dim p As Person = New Person("Tony", "Allen", 32)
Console.WriteLine(p)
// C#
Person p = new Person("Tony", "Allen", 32);
Console.WriteLine(p);
```

7. Run the console application to verify that it works correctly.

Exercise 2: Add an Enumeration to a Structure

In this exercise, you will extend the structure you created in Exercise 1 by adding an enumeration.

1. Open the project you created in Exercise 1.

2. Declare a new enumeration in the *Person* structure. Name the enumeration *Genders*, and specify two possible values: *Male* and *Female*. The following code sample demonstrates this:

```
' VB
Enum Genders
Male
Female
End Enum
// C#
public enum Genders : int { Male, Female };
```

3. Add a public member of type *Genders*, and modify the *Person* constructor to accept an instance of *Gender*. The following code demonstrates this:

```
' VB
Public firstName As String
Public lastName As String
Public age As Integer
Public gender As Genders
Public Sub New(ByVal _firstName As String, ByVal _lastName As
String, _
ByVal _age As Integer, ByVal _gender As Genders)
     firstName = _firstName
     lastName = _lastName
     age = _age
     gender = _gender
End Sub
// C#
public string firstName;
```

```
public string lastName;
public int age;
public Genders gender;
public Person(string _firstName, string _lastName, int _age,
Genders _gender)
{
     firstName = _firstName;
     lastName = _lastName;
     age = _age;
     gender = _gender;
```

4. Modify the *Person.ToString* method to also display the gender, as the following code sample demonstrates:

```
' VB
Public Overloads Overrides Function ToString() As String
Return firstName + " " + lastName + " (" + gender.ToString() +
"), age " +
age.ToString
End Function
// C#
public override string ToString()
{
     return firstName + " " + lastName + " (" + gender + "),
age " + age;
}
```

5. Modify your *Main* code to properly construct an instance of the *Person* class, as

the following code sample demonstrates:

```
' VB
Sub Main()
Dim p As Person = New Person("Tony", "Allen", 32,
Person.Genders.Male)
Console.WriteLine(p)
End Sub
// C#
static void Main(string[] args)
{
     Person p = new Person("Tony", "Allen", 32,
     Person.Genders.Male);
     Console.WriteLine(p.ToString());
}
```

6. Run the console application to verify that it works correctly.

Exercise 3: Identify Types as Value or Reference

In this exercise, you will write a console application that displays whether objects are value or reference types.

1. Using Visual Studio, create a new console application project. Name the project **List-Value-Types**.

2. Create instances of the following classes:

- *SByte*
- *Byte*

- *Int16*
- *Int32*
- *Int64*
- *String*
- *Exception*

The following code demonstrates this:

```
' VB
Dim a As SByte = 0
Dim b As Byte = 0
Dim c As Int16 = 0
Dim d As Int32 = 0
Dim e As Int64 = 0
Dim s As String = ""
Dim ex As Exception = New Exception
// C#
SByte a = 0;
Byte b = 0;
Int16 c = 0;
Int32 d = 0;
Int64 e = 0;
string s = "";
```

Exception ex = new Exception();

3. Add each of the instances to a new object array, as the following code demonstrates:

```
' VB
Dim types As Object() = {a, b, c, d, e, s, ex}
// C#
object[] types = { a, b, c, d, e, s, ex };
```

4. Within a *foreach* loop, check the *object.GetType().IsValueType* property to determine whether the type is a value type. Display each type name and whether it is a value type or a reference type, as the following code demonstrates:

```
' VB
For Each o As Object In types
Dim type As String
If o.GetType.IsValueType Then
     type = "Value type"
Else
     type = "Reference Type"
End If
     Console.WriteLine("{0}: {1}", o.GetType, type)
Next
// C#
foreach ( object o in types )
{
     string type;
     if (o.GetType().IsValueType)
          type = "Value type";
     else
          type = "Reference Type";
     Console.WriteLine("{0}: {1}", o.GetType(), type );
```

```
}
```

5. Run the console application, and verify that each type matches your understanding.

Exercise 4: Work with Strings and Arrays

In this exercise, you will write a function to sort a string.

1. Using Visual Studio, create a new console application project. Name the project **SortString**.

2. Define a string. Then use the *String.Split* method to separate the string into an array of words. The following code demonstrates this:

```
' VB
Dim s As String = "Microsoft .NET Framework 2.0
Application Development Foundation"
Dim sa As String() = s.Split(" ")
// C#
string s = "Microsoft .NET Framework 2.0 Application
Development Foundation";
string[] sa = s.Split(' ');
```

3. Call the *Array.Sort* method to sort the array of words, as the following code demonstrates:

```
' VB
Array.Sort(sa)
// C#
Array.Sort(sa);
```

4. Call the *String.Join* method to convert the array of words back into a single string, and then write the string to the console. The following code sample demonstrates this:

```
' VB
s = String.Join(" ", sa)
Console.WriteLine(s)
// C#
s = string.Join(" ", sa);
Console.WriteLine(s);
```

5. Run the console application, and verify that it works correctly.

Exercise 5: Work with Streams and Exceptions

Consider a scenario in which a coworker wrote a simple Windows Forms application to view text files. However, users complain that it is very temperamental. If the user mistypes the filename or if the file is not available for any reason, the application fails
with an unhandled exception error. You must add exception handling to the application
to display friendly error messages to users if a file is not available.

1. Copy the Chapter01\Lesson2-ViewFile folder from the companion CD to your hard disk, and open either the C# version or the Visual Basic .NET version of the ViewFile project.

2. Exceptions occur when users attempt to view a file. Therefore, edit the code that runs for the *showButton.Click* event. Add code to catch any type of exception that occurs, and display the error in a dialog box to the user. If an exception occurs after

the *TextReader* object is initialized, you should close it whether or not an exception occurs. You will need two nested *Try* blocks: one to catch exceptions during the *TextReader* initialization, and a second one to catch exceptions when the file is read. The following code sample demonstrates this:

```
' VB
Try
Dim tr As TextReader = New
StreamReader(locationTextBox.Text)
Try
displayTextBox.Text = tr.ReadToEnd
Catch ex As Exception
MessageBox.Show(ex.Message)
Finally
tr.Close()
End Try
Catch ex As Exception
     MessageBox.Show(ex.Message)
End Try
// C#
try
{
TextReader tr = new StreamReader(locationTextBox.Text);
try
{ displayTextBox.Text = tr.ReadToEnd(); }
catch (Exception ex)
{ MessageBox.Show(ex.Message); }
finally
{ tr.Close(); }
}
catch (Exception ex)
{ MessageBox.Show(ex.Message); }
```

3. Run your application. First verify that it can successfully display a text file. Then provide an invalid filename, and verify that a message box appears when an invalid filename is provided.

4. Next add overloaded exception handling to catch *System.IO.FileNotFoundException* and *System.UnauthorizedAccessException*. The following code sample demonstrates this:

```
' VB
Try
Dim tr As TextReader = New
StreamReader(locationTextBox.Text)
Try
displayTextBox.Text = tr.ReadToEnd
Catch ex As Exception
MessageBox.Show(ex.Message)
Finally
tr.Close()
End Try
Catch ex As System.IO.FileNotFoundException
MessageBox.Show("Sorry, the file does not exist.")
```

```
Catch ex As System.UnauthorizedAccessException
MessageBox.Show("Sorry, you lack sufficient privileges.")
Catch ex As Exception
MessageBox.Show(ex.Message)
End Try
// C#
try
{
     TextReader tr = new
     StreamReader(locationTextBox.Text);
try
{ displayTextBox.Text = tr.ReadToEnd(); }
catch (Exception ex)
{ MessageBox.Show(ex.Message); }
finally
{ tr.Close(); }
}
catch (System.IO.FileNotFoundException ex)
{ MessageBox.Show("Sorry, the file does not exist."); }
catch (System.UnauthorizedAccessException ex)
{ MessageBox.Show("Sorry, you lack sufficient
privileges."); }
catch (Exception ex)
{ MessageBox.Show(ex.Message); }
```

5. Run your application again, and verify that it provides your new error message if an invalid filename is provided.

```
// C#
try
{
TextReader tr = new StreamReader(locationTextBox.Text);
try
{ displayTextBox.Text = tr.ReadToEnd(); }
catch (Exception ex)
{ MessageBox.Show(ex.Message); }
finally
{ tr.Close(); }
}
catch (System.IO.FileNotFoundException ex)
{ MessageBox.Show("Sorry, the file does not exist."); }
catch (System.UnauthorizedAccessException ex)
{ MessageBox.Show("Sorry, you lack sufficient
privileges."); }
catch (Exception ex)
{ MessageBox.Show(ex.Message); }
```

5. Run your application again, and verify that it provides your new error message if an invalid filename is provided.

Exercise 6: Examine Implicit Conversion

In this exercise, you will examine conversion to determine which number types allow implicit conversion.

1. Create a new console application in Visual Studio.
2. Declare instances of three value types: *Int16*, *Int32*, and *double*. The following code sample demonstrates this:

```
' VB
Dim i16 As Int16 = 1
Dim i32 As Int32 = 1
Dim db As Decimal = 1
// C#
Int16 i16 = 1;
Int32 i32 = 1;
double db = 1;
```

3. Attempt to assign each variable to all the others, as the following code sample demonstrates.

```
' VB
i16 = i32
i16 = db
i32 = i16
i32 = db
db = i16
db = i32
// C#
i16 = i32;
i16 = db;
i32 = i16;
i32 = db;
db = i16;
db = i32;
```

4. Attempt to build your project. Which implicit conversions did the compiler allow, and why?

Exercise 7: Enumerating Through the Files in a Folder
In this exercise, you will enumerate through all the files in a particular drive.
1. Create a new console application named ShowFilesDemo.
2. Add an *Import* (or an *include* in C#) for the *System.IO* namespace into the new project.
3. Add a new method that takes a *DirectoryInfo* object named *ShowDirectory*.
4. Within your new method, iterate through each of the files in your directory and show them in the console one at a time. Your code might look something like this:

```
' VB
Sub ShowDirectory(ByVal dir As DirectoryInfo)
' Show Each File
Dim file As FileInfo
For Each file In dir.GetFiles()
     Console.WriteLine("File: {0}", file.FullName)
Next
End Sub
// C#
static void ShowDirectory(DirectoryInfo dir)
{
// Show Each File
```

```
foreach (FileInfo file in dir.GetFiles())
     {
     Console.WriteLine("File: {0}", file.FullName);
     }
}
```

Within the *ShowDirectory* method, iterate through each subdirectory and call the *ShowDirectory* method. Doing this will call the *ShowDirectory* recursively to find all the files for every directory. This code might look something like this:

```
' VB
' Go through subdirectories
' recursively
Dim subDir As DirectoryInfo
For Each subDir In dir.GetDirectories()
ShowDirectory(subDir)
Next
// C#
// Go through subdirectories
// recursively
foreach (DirectoryInfo subDir in dir.GetDirectories())
{
     ShowDirectory(subDir);
}
```

6. In the *Main* method, write code to create a new instance of a *DirectoryInfo* object for the Windows directory and use it to call the new *ShowDirectory* method. For example, the following code would work:

```
' VB
Dim dir As DirectoryInfo = New
DirectoryInfo(Environment.SystemDirectory)
ShowDirectory(dir)
// C#
DirectoryInfo dir = new
DirectoryInfo(Environment.SystemDirectory);
ShowDirectory(dir);
```

7. Build the project and resolve any errors. Verify that the console application successfully
lists all the files in the system directory (*Environment.SystemDirectory*).

Exercise 8: Watch for Changes in the File System

In this exercise, you will watch the file system for changes in all files that end with the
.ini extension.

1. Create a new console application named FileWatchingDemo.

2. Import the *System.IO* namespace into the new file.

3. Create a new instance of the *FileSystemWatcher* class, specifying the system directory.

For example, you could use the following code:

```
' VB
Dim watcher As New FileSystemWatcher(Environment.SystemDirectory)
```

```
// C#
FileSystemWatcher watcher =
new FileSystemWatcher(Environment.SystemDirectory);
```

Modify properties of the file system watcher to look only for .ini files, search through all subdirectories, and accept changes only if the attributes of the file change or if the file size changes. Your code might look like this:

```
' VB
watcher.Filter = "*.ini"
watcher.IncludeSubdirectories = True
watcher.NotifyFilter = _
NotifyFilters.Attributes Or NotifyFilters.Size
// C#
watcher.Filter = "*.ini";
watcher.IncludeSubdirectories = true;
watcher.NotifyFilter =
NotifyFilters.Attributes | NotifyFilters.Size;
```

5. To see the changes, add a handler for the *Changed* event of your watcher object. For example, you could use the following code:

```
' VB
AddHandler watcher.Changed, _
New FileSystemEventHandler(AddressOf watcher_Changed)
// C#
watcher.Changed +=
new FileSystemEventHandler(watcher_Changed);
```

6. Next you need the method that the *Changed* event is going to call. Inside this method, write out to the console the name of the changed file. Your code might look something like this:

```
' VB
Sub watcher_Changed(ByVal sender As Object, _
ByVal e As FileSystemEventArgs)
Console.WriteLine("Changed: {0}", e.FullPath)
End Sub
// C#
static void watcher_Changed(object sender,
FileSystemEventArgs e)
{
Console.WriteLine("Changed: {0}", e.FullPath);
}
```

7. Set the *EnablingRaisingEvents* property to *true* to tell the watcher object to start throwing events. Build the project and resolve any errors. Verify that the console application successfully reports when the attributes of any .ini file change or when the file size changes.

Exercise 9: Writing to a New File

In this exercise, you will create a new file and insert some text into it.

1. Create a new console application named FileDemo.

2. Add an *Import* (or an *include* in C#) for the *System.IO* namespace into the new project.

3. In the *Main* method, create a new *StreamWriter* from the *Create* method of the

File class.
4. Write some lines to the stream writer using the *WriteLine* method.
5. Close the *StreamWriter*. The code might look something like this when you are done:

```
' VB
Shared Sub Main(ByVal args() As String)
Dim writer As StreamWriter = _
     File.CreateText("c:\newfile.txt")
     writer.WriteLine("This is my new file")
     writer.WriteLine("Do you like its format?")
     writer.Close()
End Sub
// C#
static void Main(string[] args)
{
     StreamWriter writer =
     File.CreateText(@"c:\newfile.txt");
     writer.WriteLine("This is my new file");
     writer.WriteLine("Do you like its format?");
     writer.Close();
}
```

6. Build the project and resolve any errors. Verify that the console application creates the file by manually checking the file in the file system.

Exercise 10: Reading a File

In this exercise, you will open the file you created in Exercise 9 and show the contents in the console.
1. Open the FileDemo project you created in Exercise 1.
2. In the *Main* method after the *StreamWriter* class is closed, open the file using the *OpenText* method of the *File* class to create a new *StreamReader* object. **3.** Create a new string named *contents* and call the *ReadToEnd* method of the *StreamReader* class to get the entire contents of the file.
4. Close the *StreamReader* object.
5. Write the string to the console. Your code might look something like this:

```
' VB
Dim reader As StreamReader = _
File.OpenText("c:\newfile.txt")
Dim contents As String = reader.ReadToEnd()
reader.Close()
Console.WriteLine(contents)
// C#
StreamReader reader =
File.OpenText(@"c:\newfile.txt");
string contents = reader.ReadToEnd();
reader.Close();
Console.WriteLine(contents);
```

6. Build the project, and resolve any errors. Verify that the console application successfully shows the contents of the file in the console window.

Exercise 11: Compressing an Existing File

In this exercise, you will compress an existing file into a new compressed file.

1. Create a new console application named CompressionDemo.

2. Add an *Import* (or an *include* in C#) for the *System.IO* and *System.IO.Compression* namespaces into the new project.

3. Create a new static method (or a shared one for Visual Basic) named *Compress-File* that takes two strings: *inFilename* and *outFilename*. The method signature should look something like this:

```
' VB
Shared Sub CompressFile(ByVal inFilename As String,
_
ByVal outFilename As String)
End Sub
// C#
static void CompressFile(string inFilename,
string outFilename)
{
}
```

4. Inside this method, open a *FileStream* object (named *sourceFile*) by opening the file specified in the *inFilename*.

5. Create a new *FileStream* object (named *destFile*) by creating a new file specified in
the *outFilename*.

6. Create a new *GZipStream* object (named *compStream*), specifying the *destFile* as the stream to write the compressed data to. Also specify that this will be a compression stream. Your code might look something like this:

```
' VB
Dim compStream As _
New GZipStream(destFile, CompressionMode.Compress)
// C#
GZipStream compStream =
new GZipStream(destFile, CompressionMode.Compress);
```

7. Stream the data in the source file into the compression stream one byte at a time. Your code might look something like this:

```
' VB
Dim theByte As Integer = sourceFile.ReadByte()
While theByte <> -1
     compStream.WriteByte(CType(theByte, Byte))
     theByte = sourceFile.ReadByte()
End While
// C#
int theByte = sourceFile.ReadByte();
while (theByte != -1)
{
     compStream.WriteByte((byte)theByte);
     theByte = sourceFile.ReadByte();
}
```

8. Close the all the streams before exiting the method.

9. In the *Main* method of the new console project, call the *CompressFile* method with an existing file and a new file name (typically ending the source file with *.gz*). The call might look something like this:

```
' VB
CompressFile("c:\boot.ini", "c:\boot.ini.gz")
// C#
CompressFile(@"c:\boot.ini", @"c:\boot.ini.gz");
```

10. Build the project and resolve any errors. Verify that the console application created
the new compressed file by manually checking the compressed file in the
file system. The file should read as gibberish in a text editor (such as NotePad).

Exercise 12: Uncompressing the New File

In this exercise, you will open the file you created in Exercise 11 and uncompress the file into a new file.

1. Open the CompressionDemo project you created in Exercise 11.

2. Create a new static method (or a shared one for Visual Basic) named *Uncompress-File* that takes two strings: *inFileName* and *outFileName*. The method signature should look something like this:

```
' VB
Shared Sub UncompressFile(ByVal inFilename As String, _
ByVal outFilename As String)
End Sub
// C#
static void UncompressFile(string inFilename,
string outFilename)
{
}
```

3. Inside this method, open a *FileStream* object (named *sourceFile*) by opening the file specified in the *inFilename*, which will be the compressed file you wrote in Exercise 1.

Create a new *FileStream* object (named *destFile*) by creating a new file specified in the *outFilename*.

5. Create a new *GZipStream* object (named *compStream*), specifying the *sourceFile* as the stream to read the compressed data from. Also specify that this will be a decompression stream. Your code might look something like this:

```
' VB
Dim compStream As _
New GZipStream(sourceFile, CompressionMode.Decompress)
End Sub
// C#
GZipStream compStream =
new GZipStream(sourceFile, CompressionMode.Decompress);
```

6. Stream the data in the compression file into the destination file one byte at a time. Your code might look something like this:

```
' VB
Dim theByte As Integer = compStream.ReadByte()
While theByte <> -1
```

```
        destFile.WriteByte(CType(theByte, Byte))
        theByte = compStream.ReadByte()
    End While
    // C#
    int theByte = compStream.ReadByte();
    while (theByte != -1)
    {
        destFile.WriteByte((byte)theByte);
        theByte = compStream.ReadByte();
    }
```

7. Close the all the streams before exiting the method.

8. In the *Main* method of the new console project, call the *UncompressFile* method and pass it the file name of the compressed file you created in Exercise 1 and the name of a file that will receive the uncompressed data. The call might look something like this:

```
        ' VB
        DecompressFile("c:\boot.ini.gz", "c:\boot.ini.test")
        // C#
        DecompressFile(@"c:\boot.ini.gz",
    @"c:\boot.ini.test");
```

9. Build the project and resolve any errors. Verify that the console application creates the new uncompressed file by opening it with NotePad. Compare the file's contents to the original file to see whether they are identical.

Exercise 13: Creating a File in Isolated Storage

In this exercise, you will create a new file in isolated storage.

1. Create a new console application named IsolatedStorageDemo.

2. Add an *Import* (or an *include* in C#) for the *System.IO* and *System.IO.Isolated-Storage* namespaces into the new project.

3. In the *Main* method of the new project, create a new instance of the *IsolatedStorageFile*
object named *userStore* that is scoped to the current user and assembly.
Your code might look something like this:

```
    ' VB
    IsolatedStorageFile userStore = _
    IsolatedStorageFile.GetUserStoreForAssembly()
    // C#
    IsolatedStorageFile userStore =
    IsolatedStorageFile.GetUserStoreForAssembly();
```

4. Create a new instance of the *IsolatedStorageFileStream* object, passing in the name *UserSettings.set* and the new store, as shown in this example:

```
    ' VB
    IsolatedStorageFileStream userStream = new _
    IsolatedStorageFileStream("UserSettings.set", _
    FileMode.Create, _
    userStore)
    // C#
    IsolatedStorageFileStream userStream = new
    IsolatedStorageFileStream("UserSettings.set",
```

```
        FileMode.Create,
        userStore);
```

5. Use a *StreamWriter* to write some data into the new stream, and close the writer when finished. Your code might look something like this:

```
        ' VB
        StreamWriter userWriter = new StreamWriter(userStream)
        userWriter.WriteLine("User Prefs")
        userWriter.Close()
        // C#
        StreamWriter userWriter = new StreamWriter(userStream);
        userWriter.WriteLine("User Prefs");
        userWriter.Close();
```

6. Build the project and resolve any errors. Verify that the console application created the new compressed file by checking the file in the file system. The file will exist in a directory under C:\Documents and Settings\<*user*>\Local Settings\ Application Data\IsolatedStorage. This directory is a cache directory, so you will find some machine-generated directory names. You should find the files if you dig deeper into the AssemFiles directory.

Exercise 14: Reading a File in Isolated Storage

In this exercise, you will read the file you created in Exercise 14.

1. Open the project you created in Exercise 1 (IsolatedStorageDemo).

2. After the code from Exercise 14, add code to check whether the file exists in the store and show a console message if no files were found. Your code might look something like this:

```
        ' VB
        Dim files() As String =
    userStore.GetFileNames("UserSettings.set")
        If files.Length = 0 Then
        Console.WriteLine("No data saved for this user")
        End If
        // C#
        string[] files =
    userStore.GetFileNames("UserSettings.set");
        if (files.Length == 0)
        {
        // ...
        }
```

3. If the file was found, create a new *IsolatedStorageFileStream* object that opens the file you created in Exercise 1. Create a *StreamReader* to read all the text in the file into a new local string variable. Your code might look something like this:

```
        ' VB
        userStream = New _
        IsolatedStorageFileStream("UserSettings.set",
    FileMode.Open, userStore)
        Dim userReader As StreamReader = New
    StreamReader(userStream)
        Dim contents As String = userReader.ReadToEnd()
        // C#
        userStream = new
        IsolatedStorageFileStream("UserSettings.set",
```

```
        FileMode.Open, userStore);
        StreamReader userReader = new StreamReader(userStream);
        string contents = userReader.ReadToEnd();
```

4. Add a line to the console that shows the string you created from the *StreamReader*.

5. Build the project and resolve any errors. Verify that the console application shows the contents of the file on the command line.

Exercise 15: Convert a Text File to a Different Encoding Type

In this exercise, you convert a text file to UTF-7.

1. Use Visual Studio 2005 to create a blank console application.

2. Write code to read the C:\boot.ini file, and then write it to a file named bootutf7.txt using the UTF-7 encoding type. For example, the following code (which requires the *System.IO* namespace) would work:

```
        ' VB
        Dim sr As StreamReader = New StreamReader("C:\boot.ini")
        Dim sw As StreamWriter = New StreamWriter("boot-
    utf7.txt", False, Encoding.UTF7)
        sw.WriteLine(sr.ReadToEnd)
        sw.Close()
        sr.Close()
        // C#
        StreamReader sr = new StreamReader(@"C:\boot.ini");
        StreamWriter sw = new StreamWriter("boot-utf7.txt",
    false, Encoding.UTF7);
        sw.WriteLine(sr.ReadToEnd());
        sw.Close();
        sr.Close();
```

3. Run your application, and open the boot-utf7.txt file in Notepad. If the file was translated correctly, Notepad will display it with some invalid characters because Notepad does not support the UTF-7 encoding type.

Exercise 16: Create a Collection of Strings and Sort Them

In this exercise, you will create a new console application that creates a simple collection, adds several strings, and displays them in the console window. You will then sort the collection and show the items in the collection in the console window in the new order.

1. Create a new console application called BasicCollection.

2. In the main code file, include (or import for Visual Basic) the *System.Collections* namespace.

3. In the *Main* method of the project, create a new instance of the *ArrayList* class.

4. Add four strings to the new collection, "*First*", "*Second*", "*Third*", and "*Fourth*".

5. Iterate over the collection, and show each item in the console window on a separate line.

6. Next call the *Sort* method on the collection to sort its members.

7. Iterate over the collection again, and show each item in the console window to confirm that the order is now different. Your resulting code might look something like this:

```
' VB
Imports System.Collections
Class Program
Public Overloads Shared Sub Main(ByVal args() As String)
Dim myList As New ArrayList()
myList.Add("First")
myList.Add("Second")
myList.Add("Third")
myList.Add("Fourth")
For Each item as String In myList
Console.WriteLine("Unsorted: {0}", item)
Next item
' Sort using the standard comparer
myList.Sort()
For Each item as String In myList
     Console.WriteLine(" Sorted: {0}", item)
Next item
End Sub
End Class
// C#
using System.Collections;
class Program
{
static void Main(string[] args)
{
ArrayList myList = new ArrayList();
myList.Add("First");
myList.Add("Second");
myList.Add("Third");
myList.Add("Fourth");
foreach (string item in myList)
{
     Console.WriteLine("Unsorted: {0}", item);
}
// Sort using the standard comparer
myList.Sort();
foreach (string item in myList)
{
     Console.WriteLine(" Sorted: {0}", item);
}
}
}
```

8. Build the project, and resolve any errors. Verify that the console application successfully shows items in the console window, both unsorted at first and then sorted.

Exercise 17: Create and Use a Queue

In this exercise, you will create a queue, add items to it, and empty the queue to the console window.

1. Create a new console application called SequentialCollections.

2. In the main code file, include (or import for Visual Basic) the *System.Collections* namespace.

3. In the *Main* method of the project, create a new instance of the *Queue* class.

4. Add four strings to the new collection: *"First"*, *"Second"*, *"Third"*, and *"Fourth"*.

5. Empty the queue, one item at a time, using the *Count* property to test whether the collection is empty. Your resulting code might look something like this:

```
' VB
Imports System.Collections
Class Program
Public Shared Sub Main(ByVal args() As String)
Dim queue As New Queue()
queue.Enqueue("First")
queue.Enqueue("Second")
queue.Enqueue("Third")
queue.Enqueue("Fourth")
While queue.Count > 0
     Dim obj As Object = queue.Dequeue()
     Console.WriteLine("From Queue: {0}", obj)
End While
End Sub
End Class
// C#
using System.Collections;
class Program
{
static void Main(string[] args)
{
Queue queue = new Queue();
queue.Enqueue("First");
queue.Enqueue("Second");
queue.Enqueue("Third");
queue.Enqueue("Fourth");
while (queue.Count > 0)
{
     object obj = queue.Dequeue();
     Console.WriteLine("From Queue: {0}", obj);
}
}
}
```

6. Build the project, and resolve any errors. Verify that the console application successfully runs and shows the items in the queue, in first-in, first-out order.

Exercise 18: Create and Use a Stack

In this exercise, you will create a stack, add items to it, and empty the stack to the console window.

1. Open the console application you created in Exercise 1, called Sequential-Collections.

2. After the *Queue* code, create a new instance of the *Stack* class.
3. Add four strings to the stack: *"First"*, *"Second"*, *"Third"*, and *"Fourth"*.
4. Empty the queue, one item at a time, using the *Count* property to test whether the collection is empty. Your resulting code might look something like this:

```
' VB
Dim stack As New Stack()
stack.Push("First")
stack.Push("Second")
stack.Push("Third")
stack.Push("Fourth")
While stack.Count > 0
     Dim obj As Object = stack.Pop()
     Console.WriteLine("From Stack: {0}", obj)
End While
// C#
Stack stack = new Stack();
stack.Push("First");
stack.Push("Second");
stack.Push("Third");
stack.Push("Fourth");
while (stack.Count > 0)
{
     object obj = stack.Pop();
     Console.WriteLine("From Stack: {0}", obj);
}
```

5. Build the project, and resolve any errors. Verify that the console application successfully runs and shows that the items in the stack are in reverse order of the queue's items (that is, in last-in, first-out order).

Exercise 18: Create a Lookup Table

In this exercise, you will create a lookup table for a series of numbers, parse through the digits of a string, and display the numbers in the console.
1. Create a new console application called DictionaryCollections.
2. In the main code file, include (or import for Visual Basic) the *System.Collections* namespace.
3. In the *Main* method of the project, create a new instance of the *Hashtable* class.
4. Add items into the new instance of the *Hashtable* class where the key is a string containing the numbers zero through nine, and the value is the spelled-out name of the numbers zero through nine.
5. Next create a string variable with a series of numbers in it.
6. Go through the string, one character at a time using a *foreach* construct.
7. Within the *foreach*, create a new string from the character variable you created in the *foreach* loop.
8. Check to see whether the *Hashtable* contains the key of the single character string.
9. If it does, get the value for the key from the *Hashtable* and show it in the console. Your code might look something like this:

```
' VB
```

```
Imports System.Collections
Class Program
Shared Sub Main(ByVal args() As String)
Dim lookup As Hashtable = New Hashtable()
lookup("0") = "Zero"
lookup("1") = "One"
lookup("2") = "Two"
lookup("3") = "Three"
lookup("4") = "Four"
lookup("5") = "Five"
lookup("6") = "Six"
lookup("7") = "Seven"
lookup("8") = "Eight"
lookup("9") = "Nine"
Dim ourNumber As String = "888-555-1212"
For Each c as Char In ourNumber
     Dim digit As String = c.ToString()
     If lookup.ContainsKey(digit) Then
     Console.WriteLine(lookup(digit))
End If
Next
End Sub
End Class
// C#
using System.Collections;
class Program
{
static void Main(string[] args)
{
Hashtable lookup = new Hashtable();
lookup["0"] = "Zero";
lookup["1"] = "One";
lookup["2"] = "Two";
lookup["3"] = "Three";
lookup["4"] = "Four";
lookup["5"] = "Five";
lookup["6"] = "Six";
lookup["7"] = "Seven";
lookup["8"] = "Eight";
lookup["9"] = "Nine";
string ourNumber = "888-555-1212";
foreach (char c in ourNumber)
{
     string digit = c.ToString();
     if (lookup.ContainsKey(digit))
     {
     Console.WriteLine(lookup[digit]);
     }
}
}
}
```

10. Build the project, and resolve any errors. Verify that the console application successfully spells out all the digits in the number you specified.

Exercise 19: Create a Generic Collection to Store State Data

In this exercise, you create a generic *Dictionary* to hold state abbreviations with their full names.

1. Create a new console application called GenericCollections.

2. In the *Main* method of the project, create a new instance of the generic *Dictionary* class, specifying the key to be an integer and the value to be a string.

3. Add items to the collection using country codes for the keys and country names as the values.

4. Try to add strings for the keys of the country codes to make sure that the *Dictionary* is type safe. If they fail to compile, remove them in code or comment them out.

5. Write out to the console of one of your country codes using the indexer syntax of the *Dictionary*.

6. Iterate over the collection, and write out the country code and name of the country for each *KeyValuePair* in the *Dictionary*. Your code might look something like this:

```
' VB
Class Program
Public Overloads Shared Sub Main()
Dim countryLookup As New Dictionary(Of Integer, String)()
countryLookup(44) = "United Kingdom"
countryLookup(33) = "France"
countryLookup(31) = "Netherlands"
countryLookup(55) = "Brazil"
countryLookup["64"] = "New Zealand";
Console.WriteLine("The 33 Code is for: {0}",
countryLookup(33))
     For Each item As KeyValuePair(Of Integer, String) In
countryLookup
          Dim code As Integer = item.Key
          Dim country As String = item.Value
          Console.WriteLine("Code {0} = {1}", code, country)
     Next
     End Sub
     End Class
     // C#
     class Program
     {
     static void Main(string[] args)
     {
     Dictionary<int, String> countryLookup =
     new Dictionary<int, String>();
     countryLookup[44] = "United Kingdom";
     countryLookup[33] = "France";
     countryLookup[31] = "Netherlands";
     countryLookup[55] = "Brazil";
     //countryLookup["64"] = "New Zealand";
     Console.WriteLine("The 33 Code is for: {0}",
countryLookup[33]);
     foreach (KeyValuePair<int, String> item in countryLookup)
```

```
{
     int code = item.Key;
     string country = item.Value;
     Console.WriteLine("Code {0} = {1}", code, country);
}
     Console.Read();
}
}
```

7. Build the project, and resolve any errors. Verify that the console application successfully
shows all the countries added.

Paragraph 4.Final test questions

1. Data types - Different data types (int , float, enums etc.), Value types, reference types, Nullable.
2. Comparison value types and reference types.
3. Strings: String, string builder, compare strings.
4. Store and load strings from files.
5. Exceptions - exception handler, try catch blocks, Use an example: “Failure to open a file.”
6. Convert string to the array of words. Sort array of word, Create string from an array of words.
7. Interfaces. Implementation of interfaces.
8. Inheritance. Use an example “Manager inherits from a worker”.
9. Delegates. An example of usage delegates. Timer delegate on timer event.
10. Types of data conversion.
11. Graphics elements of windows forms - button, text, progress bar, labels, etc.
12. Drawing on the form. Draw shapes and different styles of doing this.
13. Working with fonts. - Different fonts in C# and VB
14. Streams. Streams types. filestream, g-zip stream, buffer streams.
15. File system. Get file names, rename files,
16. Data structures. Stack, deque, queues.
17. Data structures. ArrayLists, sortedList, hashtables.
18. Threads in .NET - creation and join of the threads.
19. E-Mail sending.
20. Binary and text files. Save and load data to files.

Сидоров Сергей Владимирович

.Net Технологии

Учебно-методическое пособие

Государственное образовательное учреждение высшего

профессионального образования «Нижегородский государственный университет им. Н.И. Лобачевского».

603950, Нижний Новгород, пр. Гагарина, 23.

Подписано в печать 29.11.2011. Формат 60x84 1/16.

Бумага офсетная. Печать офсетная. Гарнитура Таймс.

Усл. печ. л. 3,0. Уч.-изд. л. 3,3.

Заказ № 325. Тираж 100 экз.

Отпечатано в типографии Нижегородского госуниверситета

им. Н.И. Лобачевского

603600, г. Нижний Новгород, ул. Большая Покровская, 37

Лицензия ПД № 18-0099 от 14.05.01