msdn® training

Module 6: Building .NET-based Applications with C#

Contents

Overview	1
Lesson: Examining the .NET Framework Class Library	2
Lesson: Overriding Methods from System.Object	9
Lesson: Formatting Strings and Numbers	15
Lesson: Using Streams and Files	25
Review	37
Lab 6.1: Using Streams	39

Microsoft®

Instructor Notes

Presentation:
120 minutes

Lab:
60 minutes

This module presents the Microsoft® .NET Framework class library, the Object Browser, and methods that are inherited from the **System.Object** class. This module also explains how to format strings and numbers and how to use streams and files.

After completing this module, students will be able to:

- Identify a namespace in the .NET Framework class library by its function.
- Override and implement the **ToString** method.
- Format strings, currency, and date values.
- Read and write both binary and text files.

Required materials

To teach this module, you need the following materials:

- Microsoft PowerPoint® file 2609A_06.ppt
- Module 6, "Building .NET-based Applications with C#"

Preparation tasks

To prepare for this module:

- Read all of the materials for this module.
- Complete the practices and lab.
- Practice the instructor-led demonstration of the StringOrStringBuilder application.

How to Teach This Module

This section contains information that will help you to teach this module.

The practices for this module are scheduled to last approximately 10 minutes each. Encourage the students to try the practices and, if they cannot complete the steps, to open the solution file in a new instance of Microsoft Visual Studio® .NET, if a solution is available.

The practices for this module are labeled as hands-on or guided. However, you can approach the practices in the following three ways depending on your students' needs:

- If your students are experienced with the topic and most of them successfully complete the practice in the allotted time, you do not need to intervene. This strategy is called a *hands-on practice*.
- If most of your students complete the practice but you feel they could benefit from more instruction, after about 8 minutes into a practice, you can stop them and have them watch and listen as you demonstrate how to solve the tasks on your instructor computer.
- If your students are at beginner level, you can have them perform the steps simultaneously while you demonstrate the solution on your instructor computer. This strategy is called a *guided practice*.

Lesson: Examining the .NET Framework Class Library

This section describes the instructional methods for teaching certain topics in this lesson.

- Mention that the class hierarchy shown on the .NET Framework Class Library slide represents only a selection of some of the most common classes.
- Encourage students to refer to the Visual Studio.NET documentation for a complete listing of .NET Framework class library classes and their members.
- A hands-on practice follows The Object Browser topic. Best practice, however, is to demonstrate each element of the Object Browser by opening the Object Browser on your instructor computer prior to the practice.
- Explain the difference between using the Object Browser and the Class View to examine objects and projects in a solution.

Lesson: Overriding Methods from System.Object

This section describes the instructional methods for teaching certain topics in this lesson.

- Emphasize the **ToString** method, because it is the focus of the lesson and practice. However, also mention the other inherited methods: **GetHashCode**, **Equals**, and **GetType**.
- The **GetHashCode**, **Equals**, and **GetType** methods are topics in Appendix B: Advanced Topics, in Course 2609, *Introduction to C# Programming with Microsoft .NET*. You may want to point your students to the appendix, but you are not required to teach that content.

Lesson: Formatting Strings and Numbers

This section describes the instructional methods for teaching certain topics in this lesson.

- When you refer to the code in the table that appears after the methods of the **StringBuilder** class, emphasize that each row in the table builds on the state of the object **s** in the preceding row.
- You may want to show students the sample application, StringOrStringBuilder, located in the *install_folder*\Samples\StringOrStringBuilder folder. This basic application demonstrates the performance implications of using a string variable instead of an instance of the **StringBuilder** class.

To demonstrate the StringOrStringBuilder application:

1. Open the application in Visual Studio, and then run the application.

   Inform the students that the code at the top of the application window will run for the number of iterations as entered in the **Number of Iterations** box (by default 500).

2. Click **Test String**.

   While the code is running, discuss the code and why it takes so long, which can be 30 seconds or more, depending on the speed of the computer.

3. After the code runs, click **Test StringBuilder**.

   This time, the code runs very quickly. The code at the top of the application window changes to show the equivalent code using a **StringBuilder** object.

4. Click the **Compare Results** button to calculate the speed improvement accomplished by using the **StringBuilder** class.

Lesson: Using Streams and Files

This section describes the instructional methods for teaching certain topics in this lesson.

A guided practice, Using File System Information, concludes this lesson. Use your instructor computer to lead the students through the steps that are outlined in the practice.

Review

You can use a discussion format to answer the questions so that everyone gets the benefit of knowing the right answers.

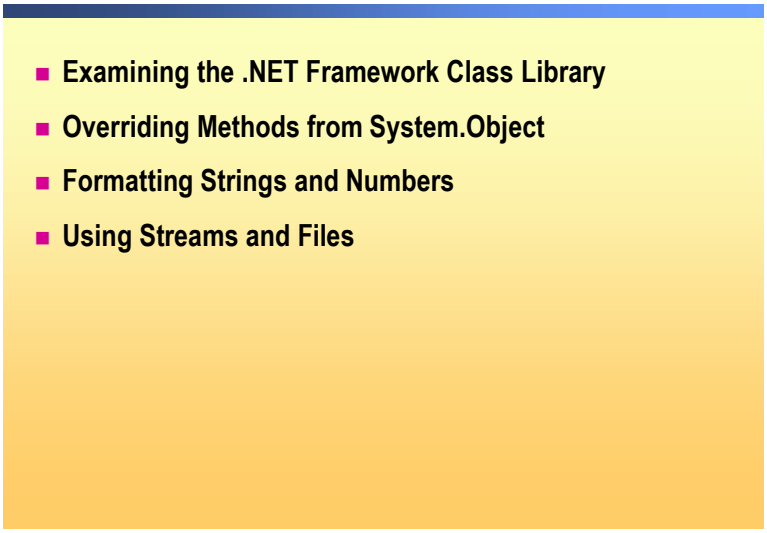
The first review question is a paper-based matching question and may not be suitable for open discussion.

Lab 6.1: Using Streams

Before beginning this lab, students should have completed all of the practices.

The lab focuses on the stream content from the module and the formatting content from the module.

Overview



Introduction

This module presents the Microsoft® .NET Framework class library, focusing on the **System.Object** class and several of its most useful derived classes.

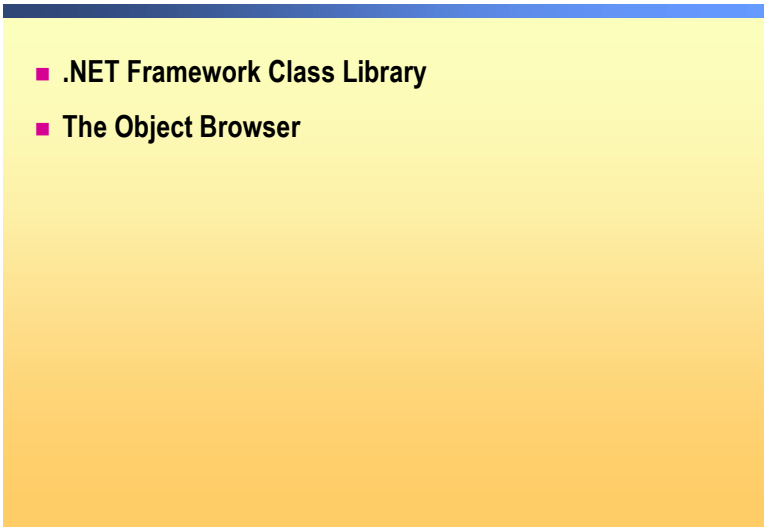
Every programming language requires access to computer features so that it can accomplish tasks such as read a file, accept input from a user, convert data types, and so on. In languages such as C and C++, an application often includes files that contain common functions so that the developer does not have to re-invent common functions. The .NET Framework class library provides common functions for all of the languages that .NET supports. This module examines some of the namespaces that convert data types, read a file, and so on, for .NET-based applications.

Objectives

After completing this module, you will be able to:

- Identify a namespace in the .NET Framework class library by its function.
- Override and implement the **ToString** method.
- Format strings, currency, and date values.
- Read and write both binary and text files.

Lesson: Examining the .NET Framework Class Library



This lesson describes the hierarchy of namespaces in the .NET Framework class library and also describes the Object Browser, the Microsoft Visual Studio® .NET feature that is used to browse the object hierarchy.

Lesson objectives

After completing this lesson, you will be able to:

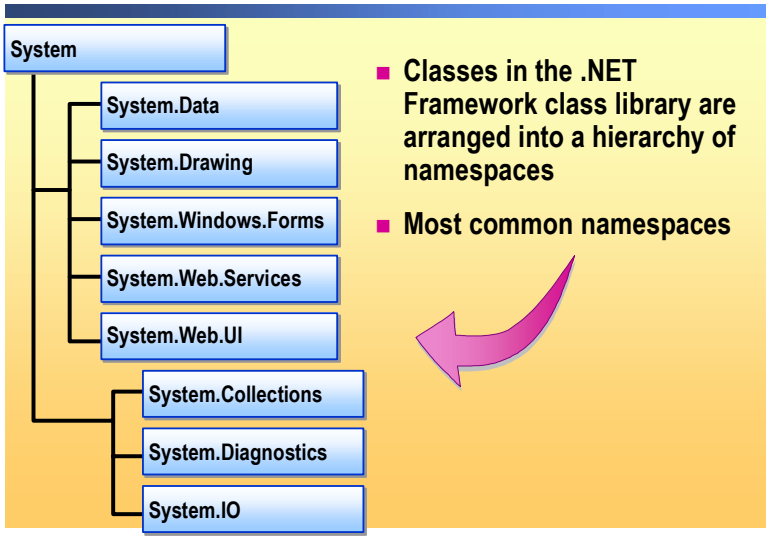
- Identify a namespace in the .NET Framework class library by its function.
- Use the Object Browser.

Lesson agenda

This lesson includes the following topics and activity:

- .NET Framework Class Library
- The Object Browser
- Practice: Using the Object Browser

.NET Framework Class Library



Introduction

Classes in the .NET Framework class library are arranged into a hierarchy of namespaces. For example, all of the classes for data collection management are in the **System.Collections** namespace.

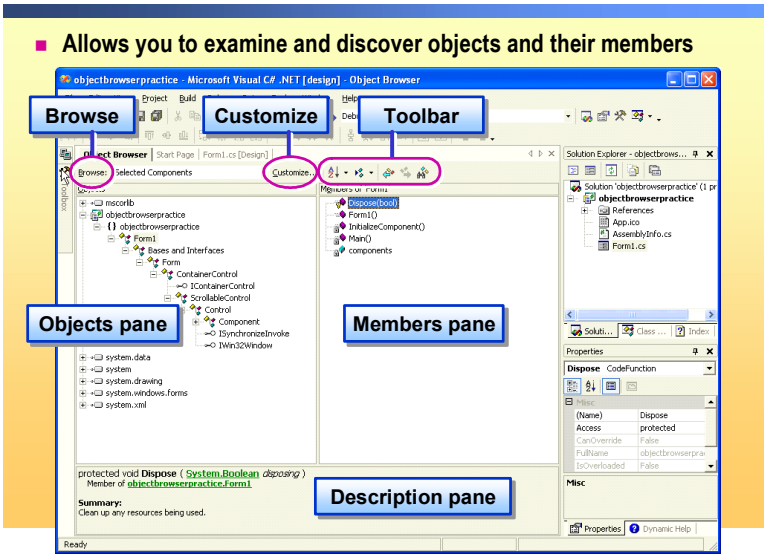
Tip For more information about the namespaces and classes in the .NET Framework class library, see the Visual Studio .NET documentation. Use the Help index and look for **Class Library**.

Common .NET Framework class library namespaces

Some of the most common namespaces in the .NET Framework class library are described in the following table.

Namespace	Description
System	Contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
System.Data	Contains most of the classes that constitute the Microsoft ADO.NET architecture. The ADO.NET architecture enables you to build components that manage data from multiple data sources.
System.Drawing	Provides access to the Graphical Device Interface (GDI+) functions. More advanced functions are provided in the **System.Drawing.Drawing2D**, **System.Drawing.Text**, and **System.Drawing.Imaging** namespaces. GDI+ is the set of classes that you use to produce any sort of drawing, graph, or image.
System.Windows.Forms	Contains classes for creating applications based on Microsoft Windows®.
System.Web.Services	Contains the classes that you use to build and use XML Web Services.
System.Web.UI	Contains classes and interfaces that allow you to create controls and pages that will appear in your Web applications as user interface on a Web page.
System.Collections	Contains interfaces and classes that define various collections of objects, such as **lists**, **queues**, **bitarrays**, **hash tables**, and **dictionaries**.
System.Diagnostics	Contains classes that allow you to interact with system processes, event logs, and performance counters. This namespace also provides classes that allow you to debug your application and to trace the execution of your code.
System.IO	Contains types that allow you to read and write files.

The Object Browser



Introduction

Use the Visual Studio .NET Object Browser to examine objects such as namespaces, classes, structures, interfaces, types, and object members such as properties, methods, events, and constants from projects in your solution, referenced components within those projects, and external components.

Tip To open the Object Browser by using shortcut keys, press CTRL+ALT+J.

Object Browser elements

Elements of the Object Browser are described in the following table.

Element	Description
Objects pane	Namespaces and their members are displayed in the Objects (left) pane. As you browse objects in this pane, you can display the inheritance hierarchy that makes up a particular member.
Members pane	If an object in the Objects pane includes members such as properties, methods, events, variables, constants, and enumerated items, those members are displayed in the Members (right) pane.
Description pane	This pane displays detailed information about the currently selected object or member, such as: • Name and parent object. • Syntax, based on the current programming language. • Links to related objects and members. • Description, comments, or Help text. • Attributes. Not every object or member has all of this information. You can copy text from the Description pane to the editor window.

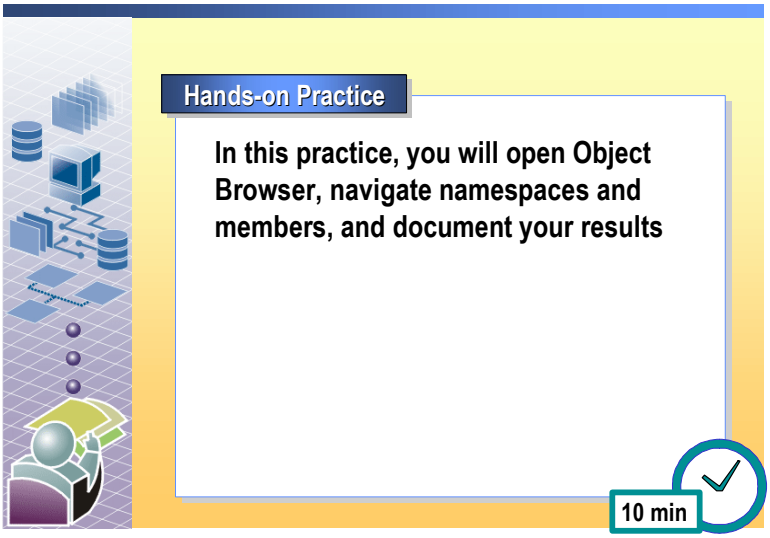
(*continued*)

Element	Description
Browse	This element allows you to locate an object within the namespace hierarchy, or to select either the **Active Project** browsing scope or the **Selected Components** browsing scope. You can determine what components are shown by choosing and customizing the browsing scope. The Active Project browsing scope is the contents of the active project and its referenced components. The Object Browser updates as the active project changes. The Selected Components browsing scope allows you to choose specific components to browse. These components can include projects in your solution and their referenced components, and any other external components, such as .NET Framework components.
Customize button	This button is available when you select **Selected Components** as your browsing scope. This displays the **Selected Components** dialog box where you specify the components that you want to browse—projects and their referenced components, and external components.
Toolbar	This element allows you to specify and customize the browsing scope, sort and group the contents of the Object Browser, move around within it, and search for symbols by using the **Find Symbol** dialog box.

Note You can also use Class View to view projects in your solution. Class View gives you a hierarchical view of symbols restricted to only the projects in your solution. You can use Class View to discover and edit the structure of your code and the relationships between objects in it.

To use Class View, in Visual Studio .NET, on the **View** menu, click **Class View**, or press CTRL+ALT+C.

Practice: Using the Object Browser



In this practice, you will open Object Browser, browse namespaces and members, and document your results.

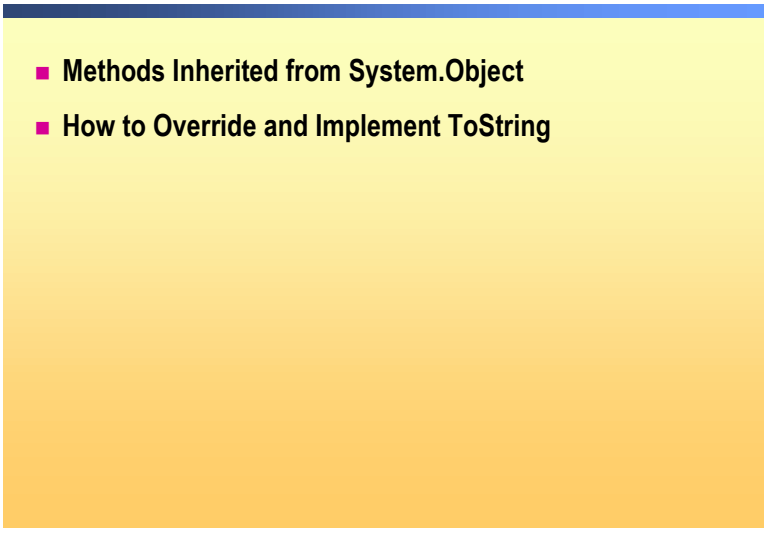
Tasks	Detailed steps
1. Start Visual Studio .NET, and then create a new project. Project Type: **Visual C#** Template: **Windows Application** Name: **ObjectBrowserPractice**	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **New Project**. **c.** In the **New Project** dialog box, under **Project Types**, click **Visual C# Projects**. **d.** Under **Templates**, click **Windows Application**. **e.** In the **Name** box, type **ObjectBrowserPractice** **f.** In the Location box, browse to *install_folder***Practices\Mod06** and then click **OK**.
2. Display the Object Browser.	▪ On the **View** menu, point to **Other Windows**, and then click **Object Browser**.

Using the Object Browser, document the **Equals** method of the **Object** object. Include the access modifiers in your documentation.

The Equals method has two forms, public static Equals(object,object) and public virtual Equals(object).

Tasks	Detailed steps
Using the Object Browser, document how many implementations of the method **Compare** are supported by the **String** object. **6**	
Using the Object Browser, find the **Convert** class and document the class modifiers that are listed for the class. In your document, include what effect the modifier has on the class. **The class modifiers for the Convert class are public and sealed. Because the class is sealed, it is not possible to derive a class from this class.**	
Using the Object Browser, find the **ReadUInt16** method. What does this method do? **Reads a 2-byte unsigned integer from the current stream using little endian encoding and advances the position of the stream by two bytes.**	
Using the Object Browser, find the **ArrayList** class. Can you set the **IsReadOnly** property to **true** or **false**? **No, the Object Browser shows this property as being a GET property as opposed to a SET GET property. You can read the value (GET) but not update the value (SET).**	
Using the Object Browser, find the **FileStream** class. What namespace contains this class? **System.IO.**	
Using the Object Browser, find the **ReadUInt32** method. What does this method do? **Reads a 4-byte unsigned integer from the current stream and advances the position of the stream by four bytes.**	

Lesson: Overriding Methods from System.Object



Introduction

Every object in the .NET Framework inherits **ToString**, **GetHashCode**, **Equals**, and **GetType** methods from **System.Object**. When you create a new object, you can override these built-in functions to improve how these functions fit your object.

Lesson objectives

After completing this lesson, you will be able to:

- Name the methods that are inherited from the **Object** class.
- Override and implement the **ToString** method.

Lesson agenda

This lesson includes the following topics and activity:

- Methods Inherited from **System.Object**
- How to Override and Implement **ToString**
- Practice: Overriding the **ToString** Method

Methods Inherited from System.Object

- **ToString**

  Creates and returns a human-readable text string that describes an instance of the class

- **GetHashCode**

  Returns an integer number as a hashcode for the object

- **Equals**

  Determines whether two objects are equal

- **GetType**

  Returns the type of the current instance

Introduction

Every object in the .NET Framework inherits from the **System.Object** base class. This class implements a small number of methods that are available on all objects. These methods are **ToString**, **GetHashCode**, **Equals**, and **GetType**. When you create a new class, the new class inherits these methods.

Overriding methods

The default implementation of **Object** class methods may not provide the function that you need for your new class, requiring you to override the method. Generally, only the **ToString**, **GetHashCode**, and **Equals** are overridden.

ToString method

The **ToString** method creates and returns a human-readable text string that describes an instance of the class.

The following code demonstrates how to call the **ToString** method:

```
object o = new object();
MessageBox.Show(o.ToString());
```

GetHashCode method

The **GetHashCode** method returns an integer number as a hash code for the object. Other .NET Framework class library classes and .NET-compatible languages such as C# use this method to quickly locate instances of an object when the object is contained in a hash table. For example, the C# statement **switch**, uses a hash table that is populated with hash entries from the **GetHashCode** method to improve the efficiency of the statement.

Equals method

The **Equals** method determines whether two objects are equal.

The following code demonstrates how to call the **Equals** method:

```
object o1 = new object();
object o2 = o1;
MessageBox.Show(o1.Equals(o2).ToString());
```

GetType method

The **GetType** method obtains the type of the current instance.

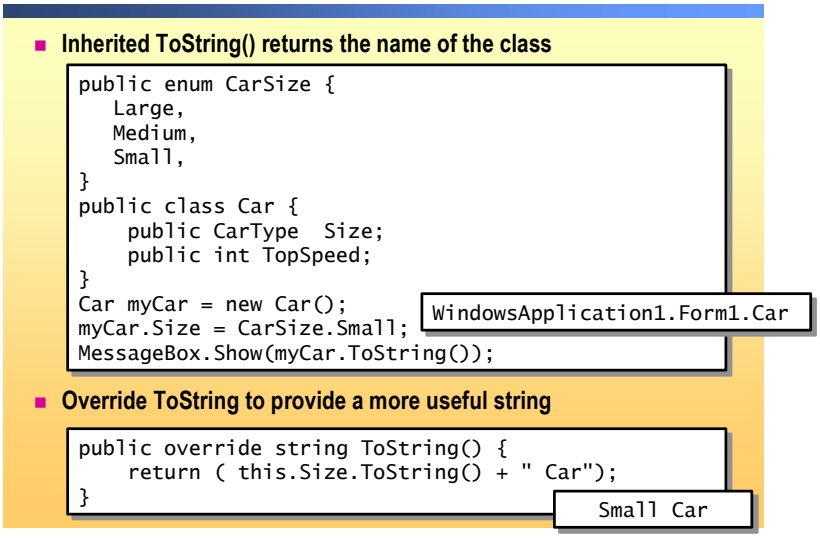
The following code demonstrates how to call the **GetType** method:

```
object o = new object();

MessageBox.Show(o.GetType().FullName);
```

It is unlikely that you would ever override the **GetType** method. It is included here because it is inherited from the **System.Object** class.

How to Override and Implement ToString



Introduction

When you create a class, the class inherits the **ToString** method.

The following code contains a **Car** class with two public fields. The code under the class writes to the console the output from the default **ToString** method that was inherited from the **System.Object** object.

Example

```
public enum CarSize {
  Large,
  Medium,
  Small,
}
public class Car {
    public CarType  Size;
    public int TopSpeed;
}
Car myCar = new Car();
myCar.Size = CarSize.Small;
MessageBox.Show(myCar.ToString());
```

The preceding line of code writes the name of the executing object to the console as follows:

```
WindowsApplication1.Form1.Car
```

Overriding the ToString method

However, if you need the **ToString** method to produce the size of the car instead of the **Car** class name, you must override the default **ToString** method, as shown in the following code:

```
public override string ToString() {
    return ( this.Size.ToString() + " Car");
}
```

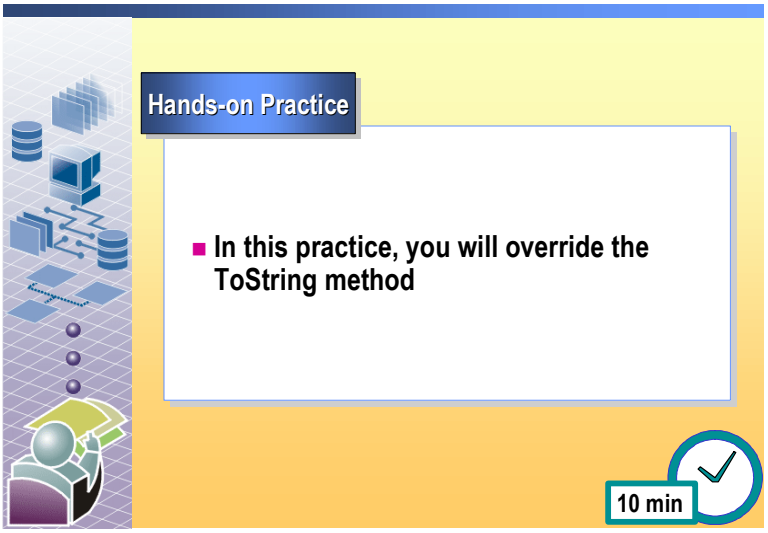
Using **ToString** on the **Car** class now produces the following output:

```
Small Car
```

This output is written to the console for the instance that was created in the preceding example.

Note For further information about overriding methods from the **System.Object** class, see Appendix B, "Advanced Topics," in Course 2609, *Introduction to C# Programming with Microsoft .NET*.

Practice: Overriding the ToString Method

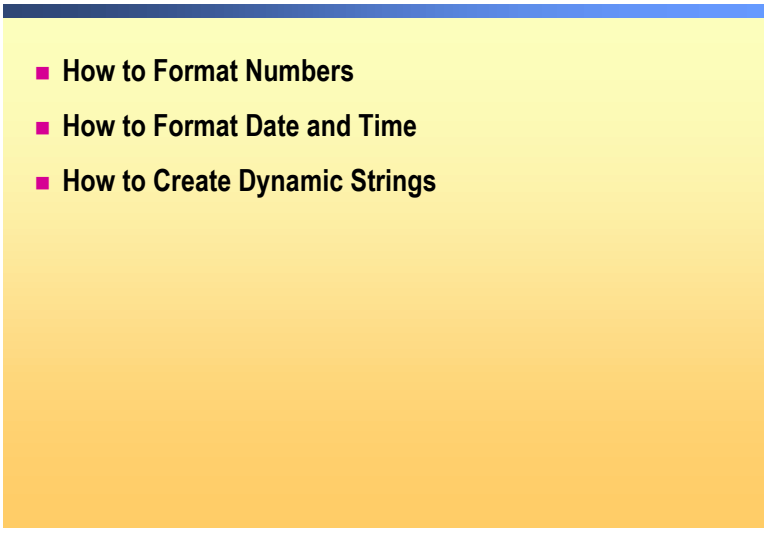


In this practice, you will override the **ToString** method.

The solution code for this practice is located in *install_folder*\Practices\Mod06\OverrideToString_Solution\OverrideToString.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open the solution *install_folder*\Practices\Mod06\OverrideToString\OverrideToString.sln	a. Start a new instance of Visual Studio .NET. b. On the **Start Page**, click **Open Project**. c. In the **Open Project** dialog box, browse to *install_folder*\Practices\Mod06\OverrideToString. d. Click **OverrideToString.sln**, and then click **Open**.
2. Run the application, and then click **Create Car Object**.	a. On the standard toolbar, click **Start**. b. In the Module 6 Practice 2 window, click **Create Car Object**.
Note: A message displays the output of the **myCar.ToString()** method. The message box contains **Mod06_Pratice2.Car**, which is the class name of the object.	
3. Override the **ToString** method to return the manufacturer and model name of the car.	▪ Refer to the content and code examples earlier in this module for detailed information about overriding the **ToString** method.
4. Run the application and then click **Create Car Object**.	a. On the standard toolbar, click **Start**. b. In the Module 6 Practice 2 window, click **Create Car Object**.
Note: Your application message box should now contain the message **BIGCARS NICECAR**.	

Lesson: Formatting Strings and Numbers



Introduction

This lesson introduces students to the classes in the .NET Framework class library that provide numeric and string formatting functions.

Lesson objectives

After completing this lesson, students will be able to:

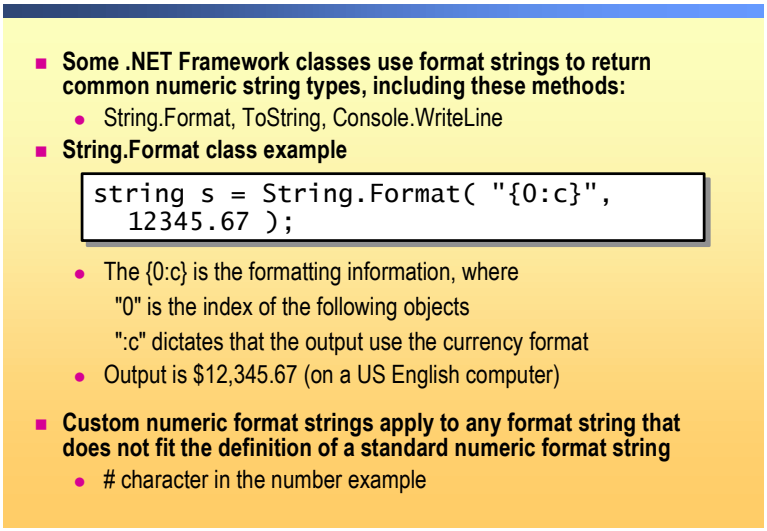
- Use the **Format** method to format numbers and currencies.
- Format currency and date values.

Lesson agenda

This lesson includes the following topics and activity:

- How to Format Numbers
- How to Format Date and Time
- How to Create Dynamic Strings
- Practice: Formatting Strings

How to Format Numbers



Introduction

Formatting refers to the various ways that you can display a particular numeric value. You use formatting to display values in a way that is appropriate for the type of application or locale.

Example of value display formats

For example, consider the value **12345.67**. You can represent this value in several ways.

You can represent the value with or without a comma:

12345.67

12,345.67

You can also display it as a negative number in various ways:

-12,345.67

(12345.67)

Finally, you can display the value by using exponential notation:

1.2E+004

1.234567E+004

The .NET Framework uses formatting strings and custom formatting strings to specify the output format of numeric values such as currency amount, fixed point digits, date, time, and so on.

A number of classes within the .NET Framework use formatting strings to specify the output format. Three examples of these methods are **String.Format**, **ToString**, and **Console.WriteLine**.

'Standard numeric format strings

A string that consists of a single alphabetic character, optionally followed by a sequence of digits that form a value between 0 and 99, is considered a *standard* format string. All other strings are considered *custom* format strings.

You use standard numeric format strings to return common numeric string types. A standard format string takes the form *Axx* where *A* is an alphabetic character that is called the *format specifier*, and *xx* is a sequence of digits that are called the *precision specifier*.

The following table describes the standard numeric format strings.

Format specifier	Name	Description
C	Currency	The number is converted to a string that represents a currency amount. The conversion is controlled by the currency format information of the **NumberFormatInfo** object that is used to format the number.
D	Decimal	This format is supported for integral types only. The number is converted to a string of decimal digits (0-9), prefixed by a minus sign if the number is negative.
E	Exponential	The number is converted to a string of the form "-*d.ddd*…E+*ddd*" or "-*d.ddd*…e+*ddd*", where each *d* indicates a digit (0-9). The string starts with a minus sign if the number is negative. One digit always precedes the decimal point. The system picks fixed point or exponential.
F	Fixed Point	The number is converted to a string of the form "-*ddd.ddd*…" where *d* is a digit (0-9). The string starts with a minus sign if the number is negative.
G	General	The number is converted to the most compact decimal form, using fixed or scientific notation.
N	Number	The number is converted to a string of the form "-*d,ddd,ddd.ddd*…", where *d* is a digit (0-9). The string starts with a minus sign if the number is negative. Thousand separators are inserted between each group of three digits to the left of the decimal point.
R	Roundtrip	The roundtrip specifier guarantees that a numeric value that is converted to a string will be parsed back into the same numeric value.
X	Hexadecimal	The number is converted to a string of hexadecimal digits. The case of the format specifier indicates whether to use uppercase or lowercase characters for the hexadecimal digits greater than 9.

Tip Use numbers after D, E, and F to control the displayed decimal places. For complete information about standard numeric format strings, see the Visual Studio .NET documentation. Use the Help index and look for **Standard Numeric Format Strings**.

Tip To maintain consistency between numeric formats and system settings, use the format codes in the preceding table rather than by creating custom formatting codes.

String.Format class example

The following code shows the **String.Format method** being used to format the numbers. "`{0:c}`" is the formatting information, where "0" is the index of the following objects and ":c" causes the number to be formatted as currency.

```
string s = String.Format( "{0:c}", 12345.67 );
```

If a string is interpreted as a standard numeric format string and contains one of the standard numeric format specifiers, the numeric value is formatted accordingly. However, if a string is interpreted as a standard format string but does not contain one of the standard format specifiers, a **FormatException** error occurs.

Custom numeric format strings

Any numeric format string that does not fit the definition of a standard numeric format string is interpreted as a *custom* numeric format string. Also, if the standard numeric format specifiers do not provide the type of formatting that you require, you can use custom format strings to further enhance string output.

The following table shows the characters that you can use to create custom numeric format strings and their definitions.

Character	Description	Example	Example output
0	Zero placeholder	{0:00#####.##}	0012345.67
#	Digit or space placeholder.	{0:#####}	12346
,	Display a comma.	{0:##,### }	12,346
.	Display the decimal point.	{0:#####.##}	12345.67
%	Display percent	{0:#%}	2%
;	Statement separator for positive, negative, and zero.	{0:##;(#);#}	The output is dependent on the input being either +, -, or 0 (zero).

Note Some of the patterns that are produced by these characters are influenced by the values in the **Regional and Language Options** settings in Control Panel.

Custom format string example

In some circumstances, you may require a number to be formatted with a number sign character (#) in the number. In the following example, the # character is a formatting character that appears at the end of the digit sequence. Typically, the # character does not appear; instead, it is interpreted as part of the formatting.

The following code demonstrates how you can use \# to cause what is called *escaping* the character. Using \# causes # character to be treated as a normal character and not part of the formatting information.

If you want to use the # character to create 123456#, use \# as follows:

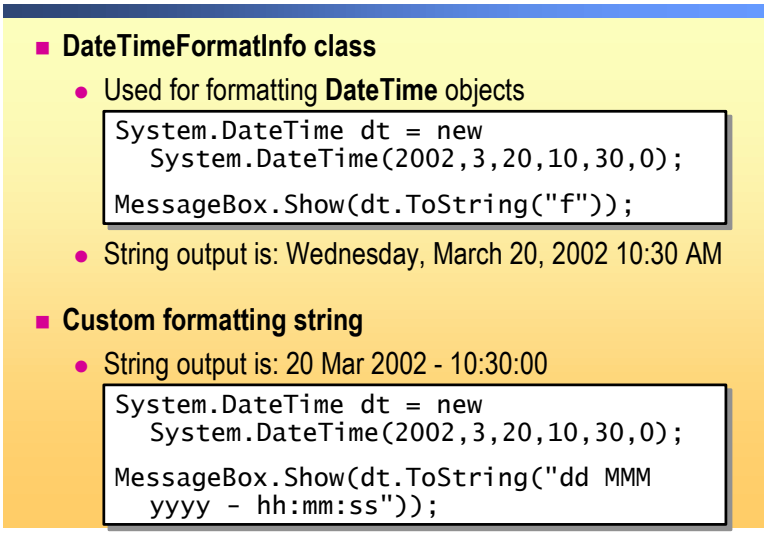
```
String.Format("{0:#\\#}",123456)
```

escape it (escape the \ to escape the #) – or better:

```
String.Format(@"{0:#\#}",123456)
```

use the verbatim string character.

How to Format Date and Time



Introduction

Like the numeric data types, the **DateTime** class implements the **IFormattable** interface, which allows you to format the value of an object as a string by using one of the overloads of the **DateTime.ToString** method. The standard format provider class that is used for formatting **DateTime** objects in the .NET Framework is **DateTimeFormatInfo**.

DateTime format string parameters are either standard format strings or custom format strings.

Tip For complete information about **DateTime** standard format strings, see the Visual Studio .NET documentation. Use the Help index and look for **Date and Time Format Strings**.

DateTime standard format strings

Format strings are interpreted as standard format specifiers if they contain only one of the single format specifiers that are listed in the following table.

Note The following table lists only a few of the most common format specifiers. For complete information about the format specifiers, see the Visual Studio .NET documentation.

Format specifier	Name	Description
d	Short date pattern	Displays a pattern defined by the **DateTimeFormatInfo.ShortDatePattern** property associated with the current thread or by a specified format provider.
D	Long date pattern	Displays a pattern defined by the **DateTimeFormatInfo.LongDatePattern** property associated with the current thread or by a specified format provider.
t	Short time pattern	Displays a pattern defined by the **DateTimeFormatInfo.ShortTimePattern** property associated with the current thread or by a specified format provider.
T	Long time pattern	Displays a pattern defined by the **DateTimeFormatInfo.LongTimePattern** property associated with the current thread or by a specified format provider.
f	Full date/time pattern (short time)	Displays a combination of the long date and short time patterns, separated by a space.
F	Full date/time pattern (long time)	Displays a pattern defined by the **DateTimeFormatInfo.FullDateTimePattern** property associated with the current thread or by a specified format provider.

Example

The following example uses the **DateTimeFormat** property.

```
System.DateTime dt = new System.DateTime(2002,3,20,10,30,0);
MessageBox.Show(dt.ToString("f"));
```

String output is:

```
Wednesday, March 20, 2002 10:30 AM
```

DateTime custom format strings

The custom format strings allow **DateTime** objects to be formatted for situations where the standard formatting strings are not useful. You can create your own custom format strings.

Tip For complete information about **DateTime** custom format strings, see the Visual Studio .NET documentation. Use the Help index and look for **Date and Time Format Strings**.

Example

The following code example uses a custom formatting string:

```
System.DateTime dt = new System.DateTime(2002,3,20,10,30,0);
MessageBox.Show(dt.ToString("dd MMM yyyy - hh:mm:ss"));
```

String output is:

```
20 Mar 2002 - 10:30:00
```

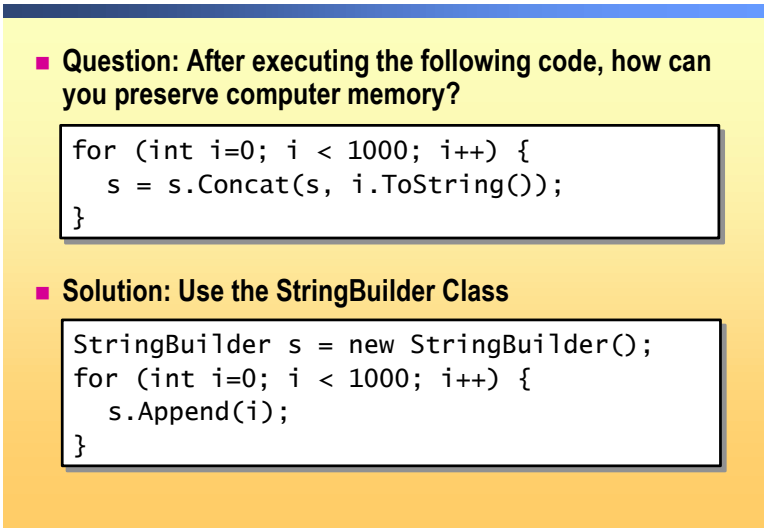
DateTime.ToString

The **DateTime.ToString** method converts the value of an instance to its equivalent string representation.

DateTime.Now

The **DateTime.Now** method returns a **DateTime** data type that is the current local date and time of the user's computer.

How to Create Dynamic Strings



Introduction

In what circumstances do you use **StringBuilder** class with a string type? Because strings are immutable, after a string is stored in memory, the memory that is allocated for the string cannot change. If the string is changed, a new memory location is needed to store the changed string. For example, consider the following:

```
int amount = 42;
string s1 = "Your balance is ";
s1 = string.Concat( s1, amount.ToString() );
```

In the preceding code, **s1** is created and then changed, causing the old and new version of **s1** to be stored temporarily in memory. The old **s1** will be cleared from memory by the garbage collection process. If your application frequently manipulates strings, you may be holding a large amount of memory in use, while waiting for the next periodic garbage collection.

Concatenating strings

The **string.Concat** method creates a new string, and concatenates **s** with the result of the **ToString** method and then stores the result in a new memory location, which is then linked to **s**. This means that you have two strings when you only need one. When dealing with multiple strings, for example if you concatenate strings in a loop, this situation can be both a performance and memory problem.

The solution is to use the **StringBuilder** class in the **System.Text** namespace.

Using the StringBuilder class

If your code must manipulate strings, especially looped operations where large numbers of strings are left in memory, it is recommended that you use the **StringBuilder** class.

StringBuilder acts just like the **Collection** classes. It allocates an initial value of sixteen characters and if your string becomes larger than this, it automatically grows to accommodate the string size. You would rewrite your code as follows:

```
int amount = 42;
StringBuilder sb = new StringBuilder( "Your balance is " );
sb.Append( amount );
```

The preceding code contains only one string, which is referenced by "sb". Also note that the **Append** method takes an object.

Methods of the StringBuilder class

The important methods of the **StringBuilder** class are listed in the following table.

Method	Function
Append	Places an item (object) at the end of the current **StringBuilder** object.
AppendFormat	Specifies a format for the object (for example, number of decimal places).
Insert	Places the object at a specific index.
Remove	Removes characters.
Replace	Replaces characters (specific or indexed).

Example

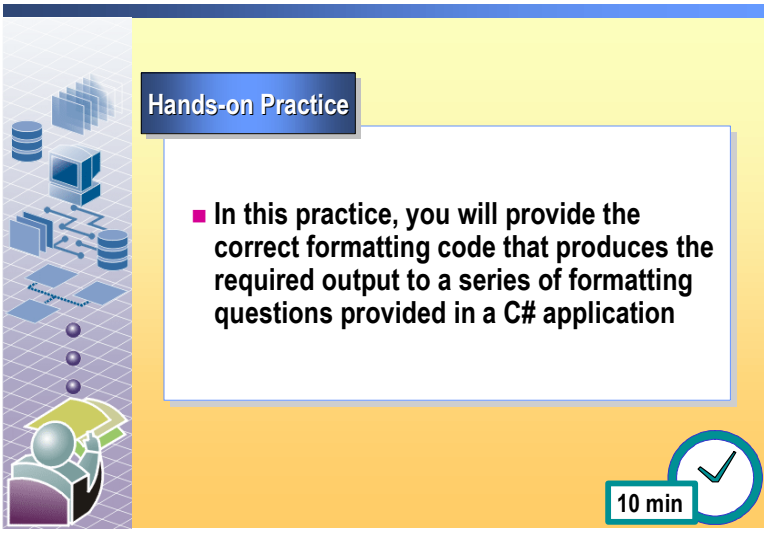
The following code shows the creation of the object **s**. The table that follows the code shows the contents of the object **s** after the code statement is executed. Each row in the table builds on the state of the object **s** from the previous row.

```
StringBuilder s = new StringBuilder("ABCD");
```

Code	Output
`s.Append("EF");`	ABCDEF
`s.AppendFormat("{0:n}",1100);`	ABCDEF1,100.00
`s.Insert(2,"Z");`	ABZCDEF1,100.00
`s.Remove(7,6);`	ABZCDEF00
`s.Replace("0","X");`	ABZCDEFXX

Note For your reference, the sample **StringOrStringBuilder** application is located at *install_folder\Samples\StringOrStringBuilder* folder on the Student Materials compact disc. This basic application demonstrates the performance implications of using a string variable instead of an instance of the **StringBuilder** class.

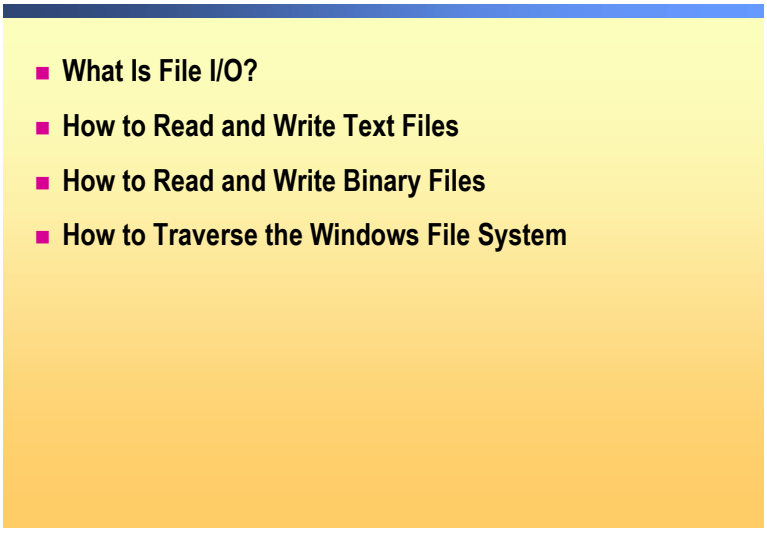
Practice: Formatting Strings



In this practice, you will provide the correct formatting code that produces the required output to a series of formatting questions that are provided in a C# application.

Tasks	Detailed steps
Note: In this practice, you are provided with an application. You do not use Visual Studio .NET for this practice.	
1. Start the application StringFormat.exe in *install_folder*\Practices\Mod06\StringFormat.	a. Click **Start**, and then click **Run**. b. Type *install_folder***Practices\Mod06\StringFormat StringFormat.exe** and then click **OK**.
2. Examine the code samples and then enter the format code that completes the code samples.	▪ Study the code samples. Enter the format code that completes the code examples.
Note: You will progress to the next question when the correct format code is entered. You can skip a question at any time by clicking **Skip**.	

Lesson: Using Streams and Files



Introduction

This lesson introduces the **System.IO** namespace and explains how to read and write binary and text files, how to browse through the file system, how to read the contents of a file, and how to write a file.

Lesson objectives

After completing this lesson, you will be able to:

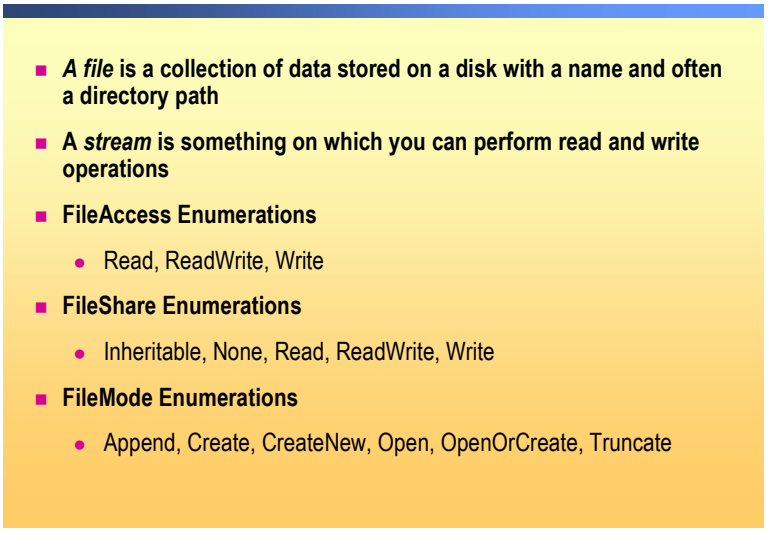
- Read and write binary files.
- Read and write text files.
- Traverse the Windows file system.

Lesson agenda

This lesson includes the following topics and activity:

- What Is File I/O?
- How to Read and Write Text Files
- How to Read and Write Binary Files
- How to Traverse the Windows File System
- Practice: Using File System Information

What Is File I/O?



Introduction

The .NET Framework distinguishes between files and streams.

A *file* is a collection of data stored on a disk with a name and often a directory path. When you open a file for reading or writing, it becomes a *stream*. You can perform read and write operations on a stream.

Streams encompass more than just open disk files, however. Data coming over a network is a stream, and you can also create a stream in memory. In a console application, keyboard input and text output are also streams.

Stream operations

Streams involve these fundamental operations:

- Streams can be read from. *Reading* is the transfer of data from a stream into a data structure, such as an array of bytes.
- Streams can be written to. *Writing* is the transfer of data from a data structure into a stream.
- Streams can support seeking. *Seeking* is the querying and modifying of the current position within a stream.

FileStream class

Most file I/O support in the .NET Framework is implemented in the **System.IO** namespace. You use the **FileStream** class in the **System.IO** namespace to read from, write to, and close files. **FileStream** inherits from the abstract class **Stream**, and many of its properties and methods are derived from **Stream**.

To open an existing file or create a new file, you create an object of type **File Stream**.

File access, sharing and type

The **FileAccess**, **FileMode**, and **FileShare** enumerations define constants that are used by some of the **FileStream** and **IsolatedStorageFileStream** constructors and some of the **File.Open** overloaded methods. These constants affect the way in which the underlying file is created, opened, and shared.

FileAccess enumeration

Unless you specify a **FileAccess** enumerator, the file is opened for both reading and writing. The **FileAccess** enumerator indicates whether you want to read from the file, write to it, or both.

FileAccess members

Members of the **FileAccess** enumeration are **Read**, **ReadWrite**, and **Write**.

The following **FileStream** constructor grants read-only access to an existing file (**FileAccess.Read**).

```
FileStream s2 = new FileStream(name, FileMode.Open,
FileAccess.Read, FileShare.Read);
```

FileShare enumerations

The **FileShare** enumerator contains constants for controlling the kind of access that other **FileStream** constructors can have to the same file. A typical use of this enumeration is to define whether two processes can simultaneously read from the same file. For example, if a file is opened and **FileShare.Read** is specified, other users can open the file for reading but not for writing.

FileShare members

Members of the **FileShare** enumerator are:

- **Inheritable**, which make the file handle inheritable by child processes.
- **None**, which declines sharing of the current file.
- **Read**, which allows subsequent opening of the file for reading.
- **ReadWrite**, which allows subsequent opening of the file for reading and writing.
- **Write**, which allows subsequent opening of the file for writing.

Example

The following **FileStream** constructor opens an existing file and grants read-only access to other users (**FileShare.Read**):

```
FileStream s2 = new FileStream(name, FileMode.Open,
  FileAccess.Read, FileShare.Read);
```

FileMode enumerations

This enumerator specifies how the operating system should open a file. A **FileMode** parameter is specified in many of the constructors for **FileStream**, **IsolatedStorageFileStream**, and in the **Open** methods of **File** and **FileInfo** to control how a file is opened.

FileMode parameters control whether a file is overwritten, created, or opened, or some combination thereof. Use **Open** to open an existing file. To append to a file, use **Append**. To truncate a file or to create it if it does not exist, use **Create**.

FileMode members

Members of **FileMode** enumerations are:

- **Append**, which opens the file if it exists and seeks to the end of the file, or creates a new file.
- **Create**, which specifies that the operating system should create a new file.
- **CreateNew**, which specifies that the operating system should create a new file.
- **Open**, which specifies that the operating system should open an existing file.
- **OpenOrCreate**, which specifies that the operating system should open a file if it exists; otherwise, a new file should be created.
- **Truncate**, which specifies that the operating system should open an existing file.

Example

The following **FileStream** constructor opens an existing file (**FileMode.Open**):

```
FileStream s2 = new FileStream(name, FileMode.Open,
   FileAccess.Read, FileShare.Read);
```

How to Read and Write Text Files

Class	Example

StreamReader	`StreamReader sr = new StreamReader(@"C:\SETUP.LOG");`
	`textBox1.Text = sr.ReadToEnd();`
	`sr.Close();`

StreamWriter	`StreamWriter sw = new StreamWriter(@"C:\TEST.LOG",false);`
	`sw.WriteLine("Log Line 1");`
	`sw.WriteLine("Log Line 2");`
	`sr.Close();`

XmlTextReader	`public class XmlTextReader : XmlReader, IXmlLineInfo`

XmlTextWriter	`w.WriteStartElement("root");`
	`w.WriteAttributeString("xmlns", "x", null, "urn:1");`
	`w.WriteStartElement("item","urn:1"); w.WriteEndElement();`
	`w.WriteStartElement("item","urn:1"); w.WriteEndElement();`
	`w.WriteEndElement();`

Introduction

Use the **Stream** class to read and write data. The **Stream** class is the abstract class that supports reading and writing bytes. If you know that your file contains only text, you may use the **StreamReader** or **StreamWriter** classes. If you know that your file contains data from different inputs, such as a stream object, a **TextReader** class object, and a URL identifying a local file location, or Web site, you may use the **XMLTextReader** and **XMLTextWriter** classes.

StreamReader class

The **StreamReader** class implements a **TextReader** that reads lines of information from a standard text file such as a log file. A **TextReader** represents a reader that can read a sequential series of characters.

The following code uses the **StreamReader** class:

```
StreamReader sr = new StreamReader(@"C:\SETUP.LOG");
textBox1.Text = sr.ReadToEnd();
```

The following line places entire file into the textbox:

```
sr.Close();
```

StreamWriter class

The **StreamWriter** class inherits from the abstract class **TextWriter** for writing characters to a stream in a particular encoding. The **TextWriter** class represents a writer that can write a sequential series of characters.

The following code uses the **StreamWriter** class:

```
StreamWriter sw = new StreamWriter(@"C:\TEST.LOG",false);
sw.WriteLine("Log Line 1");
sw.WriteLine("Log Line 2");
sr.Close();
```

XmlTextReader class

The **XmlTextReader** class inherits from the class **XmlReader**, and provides a fast, performant parser. It enforces the rules that XML must be well-formed. It is neither a validating nor a non-validating parser because it does not have a document type definition (DTD) or schema information. It can read text in blocks or read characters from a stream.

The **XmlTextReader** provides the following functionality:

- Enforces the rules that XML must be well-formed.
- Checks that the DTD is well-formed. However, **XmlTextReader** does not use the DTD for validation, expanding entity references, or adding default attributes.
- Validating is not performed against DTDs or schemas.
- Checks that any DOCTYPE nodes are well-formed.
- Checks that the entities are well-formed. For node types of **EntityReference**, a single, empty **EntityReference** node is returned. An empty **EntityReference** node is one in which its **Value** property is **string.Empty**. This is because you have no DTD or schema with which to expand the entity reference. The **XmlTextReader** does ensure that the whole DTD is well-formed, including the EntityReference nodes.
- Provides a performant XML parser, because the **XmlTextReader** does not have the overhead involved with validation checking.

The **XmlTextReader** can read data from different inputs, such as a stream object, a **TextReader Class** object, and a URL identifying a local file location or Web site.

The following code defines the **XmlTextReader** class:

```
public class XmlTextReader : XmlReader, IXmlLineInfo
```

XmlTextWriter class

The **XmlTextWriter** class represents a writer that provides a fast, non-cached, forward-only way of generating streams or files containing XML data that conforms to the World Wide Web Consortium (W3C) XML 1.0 and the namespaces in XML recommendations.

XmlTextWriter maintains a namespace stack corresponding to all of the namespaces defined in the current element stack. Using **XmlTextWriter** you can declare namespaces manually.

XmlTextWriter promotes the namespace declaration to the root element to avoid having it duplicated on the two child elements. The following code generates the XML output:

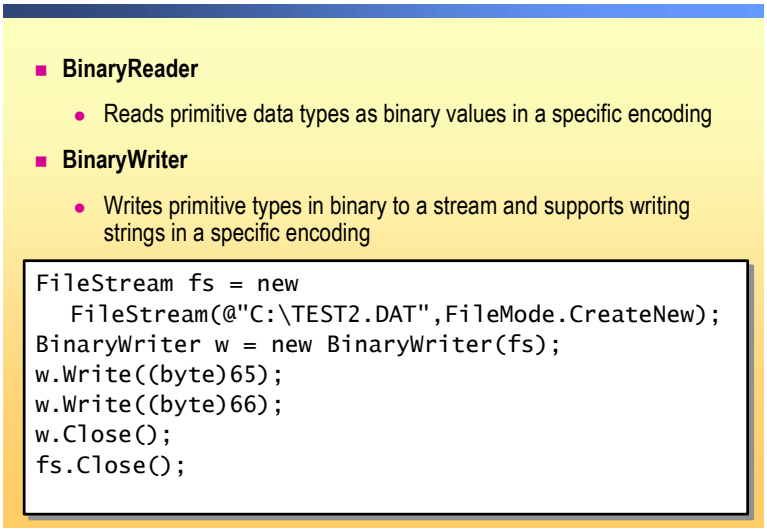
```
w.WriteStartElement("root");
w.WriteAttributeString("xmlns", "x", null, "urn:1");
w.WriteStartElement("item","urn:1"); w.WriteEndElement();
w.WriteStartElement("item","urn:1"); w.WriteEndElement();
w.WriteEndElement();
```

The child elements pick up the prefix from the namespace declaration. Given the preceding code, the code output is:

```
<root xmlns:x="urn:1">
 <x:item/>
 <x:item/>
</x:root>
```

Tip **XmlTextWriter** also allows you to override the current namespace declaration.

How to Read and Write Binary Files



Introduction

You use the **BinaryReader** and **BinaryWriter** classes for writing and reading binary data. Therefore, you use a binary rather than a text file stream when you must handle binary streams of information rather than textual information.

BinaryReader class

The **BinaryReader** class reads primitive data types as binary values in a specific encoding.

BinaryWriter class

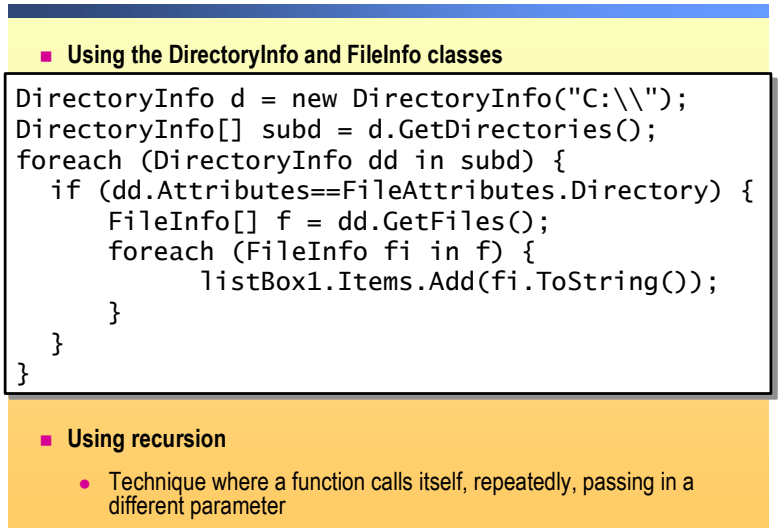
The **BinaryWriter** class writes primitive types in binary to a stream and supports writing strings in a specific encoding.

Example

The following code example demonstrates writing two bytes of data to a file:

```
FileStream fs = new
  FileStream(@"C:\TEST2.DAT",FileMode.CreateNew);
BinaryWriter w = new BinaryWriter(fs);
w.Write((byte)65);
w.Write((byte)66);
w.Close();
fs.Close();
```

How to Traverse the Windows File System



Introduction

The ability to look over files and subdirectories for a specific directory is essential for many programming tasks. You can work with files and directories by using the **DirectoryInfo** and **FileInfo** classes in combination, which is a very efficient way to obtain all of the information that you need about files and subdirectories in a specific directory.

DirectoryInfo class

The **DirectoryInfo** class exposes instance methods for creating, moving, and enumerating through directories and subdirectories. This class includes the **GetFiles** method which returns a file list from the current directory.

FileInfo class

The objects inside the directory can be files or directories. You can iterate through the directory twice, looking for files first, and directories next. An alternate solution is to use the **FileSystemInfo** object, which can represent a **FileInfo** or a **DirectoryInfo** object. Using the **FileSystemInfo** object allows you to iterate through the collection only once.

Example

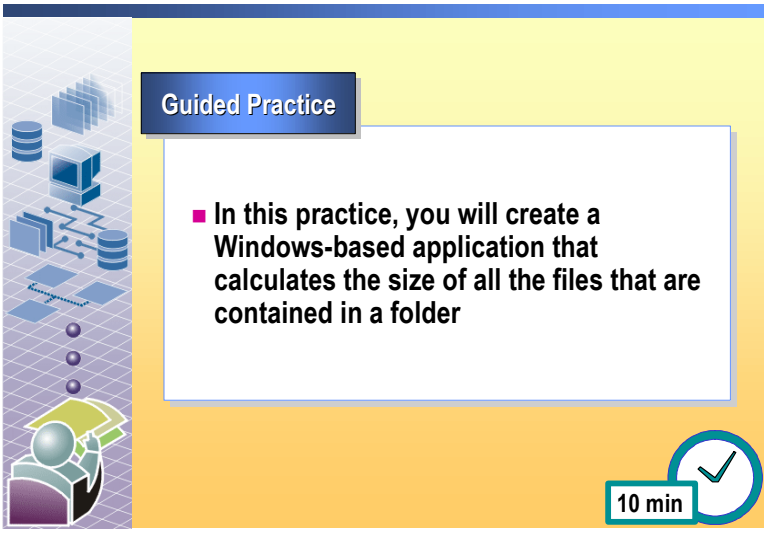
```
DirectoryInfo d = new DirectoryInfo("C:\\");
DirectoryInfo[] subd = d.GetDirectories();
foreach (DirectoryInfo dd in subd) {
  if (dd.Attributes==FileAttributes.Directory) {
      FileInfo[] f = dd.GetFiles();
      foreach (FileInfo fi in f) {
          listBox1.Items.Add(fi.ToString());
      }
  }
}
```

Using recursion

Recursion is a programming technique where a function calls itself, repeatedly, passing in a different parameter. For example, to traverse the Windows file system, you can pass the root of a particular drive, such as C:\, into a function. The function then obtains the subdirectories of this directory and calls itself for each subdirectory, and so on.

Caution Recursion can generate a **StackOverflowException** error.

Practice: Using File System Information



In this practice, you will create a Windows-based application that calculates the size of all the files that are contained in a folder.

The solution for this practice is located in *install_folder*\Practices\Mod06\Streams_Solution. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET.	▪ Start a new instance of Visual Studio .NET.
2. Open *install_folder*\Practices \Mod06\Streams\Streams.sln.	**a.** On the **Start Page**, click **Open Project**. **b.** In the **Open Project** dialog box, browse to *install_folder*\Practices \Mod06\Streams, click **Streams.sln**, and then click **Open**.
3. Open the Code Editor for Form1.	▪ In Solution Explorer, under **Solution 'Strings'**, under project **Strings**, right-click **Form1.cs**, and then click **View Code**.

Tasks	Detailed steps
4. To the **button1_Click** procedure, add code that calculates the size of the files in the directory specified in textBox1.	**a.** Scroll down through the code displayed in the window and locate the **button1_Click** procedure. **b.** Add code into this procedure that calculates the total size of the files contained in the directory specified by textBox1. Use the **MessageBox** class to display your result to the user.
5. Run the application and test it with the C:\Program Files\Msdntrain directory.	**a.** On the standard toolbar, click **Start**. **b.** In the Streams Practice window, in the **Directory** text box type **C:\Program Files\Msdntrain** and then click **Calculate Size**. The total size of the files contained in the directory should be displayed in a message box.

OPTIONAL: If you have time, try using recursion in your application to calculate the size of all the files contained in the folder specified and all subfolders of that folder.
You might want to examine the solution *install_folder*\Practices\Mod06\Streams_Recursive\Streams.sln. Notice the use of the **ref** keyword in the **calculate_size** procedure and the calls to this procedure.

Review

- Examining the .NET Framework Class Library
- Overriding Methods from System.Object
- Formatting Strings and Numbers
- Using Streams and Files

1. The following table lists namespace contents and namespaces. Draw a line to match the namespace to its contents.

Namespace	Namespace contents
System	**A**. Types that allow you to read and write files.
System.Collections	**B**. Most of the classes that constitute the ADO.NET architecture.
System.Data	**C**. Fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
System.Diagnostics	**D**. Interfaces and classes that define various collections of objects.
System.IO	**E**. Classes that allow you to interact with system processes, event logs, and performance counters.

System = C, System.Collections = D, System.Data = B, System.Diagnostics = E, System.IO = A

2. What methods are inherited from the **System.Object** base class when you create a new class?

   The ToString, GetHashCode, Equals and GetType methods are inherited from the System.Object class.

3. The **Append**, **AppendFormat**, **Insert**, and **Replace** methods belong to which class?

   The StringBuilder class.

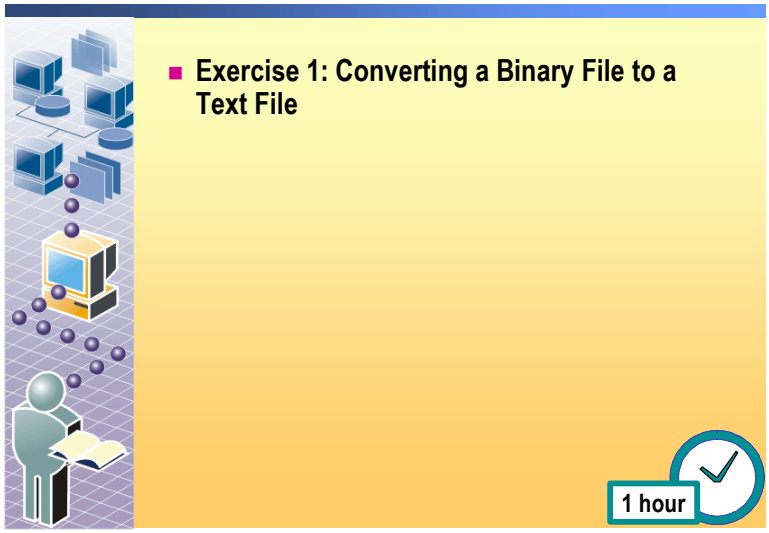
4. What type of object do you create to open an existing file or to create a new file?

   Create a FileStream object to create a new file, or to open an existing file.

5. Which two classes are used for writing and reading binary data?

   The BinaryReader and BinaryWriter classes are used for writing and reading binary data.

Lab 6.1: Using Streams



Objectives

After completing this lab, you will be able to:

- Read data from a binary file.
- Write text to a file.
- Format strings.

Prerequisites

Before working on this lab, you must have:

- Format strings, currency, and date values.
- Read and write both binary and text files.

Scenario

In this, lab you will build a C# application that will take the data contained in a binary file and write it as a human-readable text file. The data contained in the file represents transactions from a transaction clearing company. The data consists of the account number that the money was debited from, the amount of money, and the date the transaction occurred. The data is encoded in the file as shown in the following table:

B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
Account (1-999)		DayOfYear (1-365)		Year (1900-3000)		Amount Implied 2 decimal places (for example, 100.25 held as 10025)			

Your application will take the data contained in *install_folder*\Labs\Lab06_1\data.bin and convert it into a file named *install_folder*\Labs\Lab06_1\output.txt.

Your output text file should contain the columns shown in the following table:

Column	Size
Date in Long date format	44 characters
Account	3 characters
Amount shown as a currency	8 characters

Write your conversion code to calculate the total of the Amount column. Your total should equal $223,652.00.

Estimated time to complete this lab: 60 minutes

Exercise 1
Converting a Binary File to a Text File

In this exercise, you will write an application that converts a binary file to a text file.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then create a new project. Project Type: **Visual C#** Template: **Windows Application** Location: *install_folder*\Labfiles\Lab06_1 Name: **FileConversion**	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **New Project**. **c.** In the **New Project** dialog box, under **Project Types**, click **Visual C# Projects**. **d.** Under **Templates**, click **Windows Application**. **e.** In the **Location** box, type *install_folder***Labfiles\Lab06_1** **f.** In the **Name** box, type **FileConversion** and then click **OK**.
2. Write the code to convert the file.	▪ Write the necessary code to convert the file as described in the lab scenario.

Application Hints. You may wish to work through the following step guidelines to create this application.

a. Add a button to the form.

b. Add the code to for this application into the button_Click event.

c. Create a **FileStream** object to open the binary file.

d. Use the opened **FileStream** object to create a **BinaryReader** object. The **BinaryReader** object provides you with binary methods to manipulate the binary file.

e. Create a **StreamWriter** object to output the text file.

f. Add definitions for variables that your code will use.

g. Create a while loop structure to loop through the binary file. The loop condition should be:

```
(binaryReaderObj.Length > binaryReader.Position)
```

h. Within the loop, use the **ReadUInt16** method to read the 2 byte data and the **ReadUInt32** method to read the 4 byte data.

i. Calculate the running total.

j. The binary file holds the date as a year and the number of days from the beginning of the year. To use this data to set a variable of type DateTime:

```
System.DateTime dt = new System.DateTime(yearfrombinaryfile,1,1);
```

k. Use the **AddDays** method of the **DateTime** class to correct for the days in the year.

```
dt = dt.AddDays(dayofyearfrombinaryfile);
```

l. Notice above that the **AddDays** method returns a new **DateTime** object.

m. Format the data to be written into the text file. Use the **WriteLine** method to output the line of text to the text file.

n. When the loop is complete, close all stream-based objects.

3. Run the application and create the output file.	▪ Run your application.