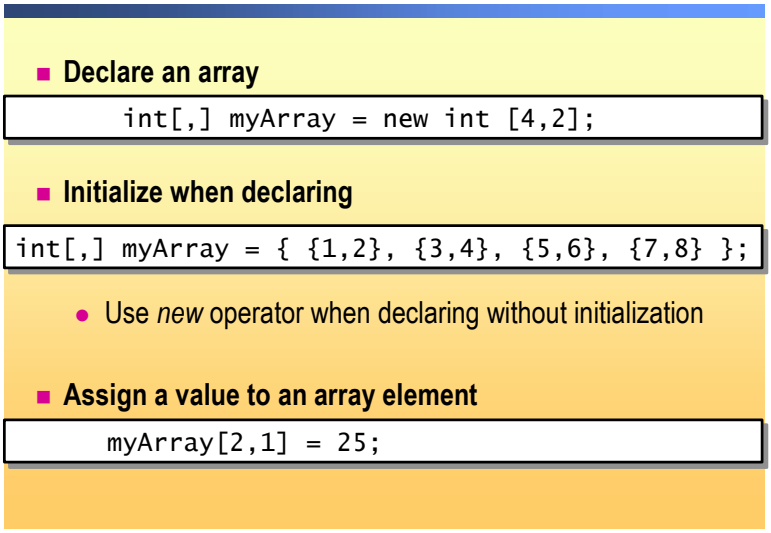
msdn® training

Appendix B: Advanced Topics

Contents

Microsoft®

How to Use Multidimensional Arrays



Introduction

You can use a multidimensional array to store values that naturally fall into a grid-like or rectangular structure. For example, to keep track of each pixel on your computer screen, you can refer to its X coordinates in one dimension and its Y coordinates in a second dimension.

Declaring

To declare a multidimensional array, use the following syntax:

```
int[,] myArray;
```

Multidimensional arrays can have any number of dimensions. The following declaration creates an array of three dimensions, 4, 2, and 3:

```
int[,,] myArray = new int [4,2,3];
```

If a two-dimensional array is like a rectangle, you can visualize a three-dimensional array as being like a cube, with the dimensions corresponding to height, width, and depth.

Initializing

You can initialize the array upon declaration, as shown in the following example:

```
int[,] myArray;
myArray = new int[,] { {1,2}, {3,4}, {5,6}, {7,8}};
```

Or more simply:

```
int[,] myArray = new int[,] {{1,2}, {3,4}, {5,6}, {7,8}};
```

Or even more simply:

```
int[,] myArray = {{1,2}, {3,4}, {5,6}, {7,8}};

string[,] siblings = new string[,] { {"Mike","Amy"},
                          {"Mary","Ray"} };
```

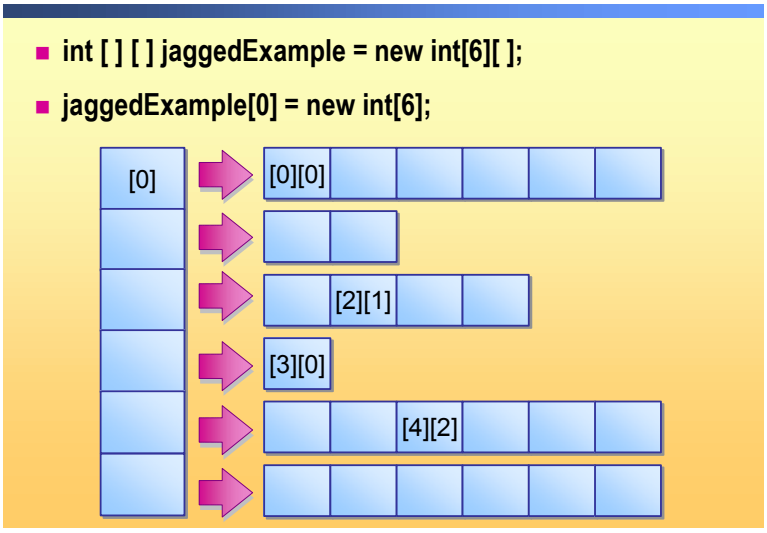
Assigning a value to an array element

You can also assign a value to an array element, for example:

```
myArray[2,1] = 25;
```

Note The total storage that the array requires increases dramatically when you start adding dimensions to an array. Declare the smallest array that you can.

How to Use Jagged Arrays



**********ILLEGAL FOR NON-TRAINER USE**********

Introduction

A *jagged array* is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes. A jagged array is sometimes called an array of arrays.

Declaring

The following code declares a two-dimensional jagged array that is made up of a single-dimensional array that has six elements, each of which is another single-dimensional array of integers:

```
int[ ][ ] myJaggedArray = new int[6][ ];
```

Initializing

Before you can use **myJaggedArray**, you must initialize its elements. You can initialize the elements as shown in the following example:

```
myJaggedArray[0] = new int[6];
myJaggedArray[1] = new int[2];
myJaggedArray[2] = new int[4];
myJaggedArray[3] = new int[1];
myJaggedArray[4] = new int[6];
myJaggedArray[5] = new int[6];
```

Each of the elements of the first array is a single-dimensional array of integers. The first element is an array of 6 integers, the second is an array of 2 integers, the third is an array of 4 integers, and so on.

Examples

You can use initializers to fill the array elements with values, in which case, you do not need the array size as shown in the following example:

```
myJaggedArray[0] = new int[] {1,3,5,7,9,11};
myJaggedArray[1] = new int[] {0,2};
myJaggedArray[2] = new int[] {11,22,33,44};
```

You can also initialize the array upon declaration as shown in the following example:

```
int[ ][ ] myJaggedArray = new int [ ][ ]
                          {
                             new int[ ] {1,3,5,7,9},
                             new int[ ] {0,2,4,6},
                             new int[ ] {11,22}
                          };
```

You can also mix jagged and multidimensional arrays. The following example shows a declaration and initialization of a single-dimensional jagged array that contains two-dimensional array elements of different sizes:

```
int[ ][,] myJaggedArray = new int [3][,]
                        {
                          new int[,] { {1,3}, {5,7} },
                          new int[,] { {0,2}, {4,6}, {8,10} },
                          new int[,] { {11,22}, {99,88}, {0,9} }
                        };
```

Accessing array elements

You can access individual array elements, as shown in the following examples:

```
// Assign 33 to the second element of the first array:
myJaggedArray[0][1] = 33;
// Assign 44 to the second element of the third array:
myJaggedArray[2][1] = 44;
```

How to Use XML with a DataSet

- Read data from a DataSet object in XML format
- Fill a DataSet object with XML data
- Create an XML Schema for the XML representation of the data in a DataSet
- Load XML data into a Document Object Model (DOM) tree, from a stream or file. You can then manipulate the data as XML or as a DataSet
- Create typed DataSets

Introduction

Microsoft® ADO.NET is tightly integrated with XML. The ADO.NET object model was designed with XML at its core. As a result, ADO.NET makes it easy to convert relational data into XML format. You can also convert data from XML into tables and relations.

Benefit of using XML

XML is a rich, platform-independent, and portable way of representing data. An important characteristic of XML data is that it is text-based. Using text-based data makes it easier to pass XML data between applications and services, even if they are running on different platforms. XML also enables organizations to exchange data without further customization of each organization's proprietary software.

Scenario

You must write an application that processes XML data. That XML data may come from an external business through an XML Web service, e-mail, Microsoft BizTalk® Server, or many other sources.

XML support

The ADO.NET object model includes extensive support for XML. Consider the following facts and guidelines when you use the XML support in ADO.NET:

- You can write data from a **DataSet** object in XML format. The XML format is useful if you want to pass data between applications or services in a distributed environment.
- You can fill a **DataSet** object with XML data. This is useful if you receive XML data from another application or service, and want to update a database by using this data.
- You can create an XML Schema for the XML representation of the data in a DataSet. You can use the XML Schema to perform tasks such as serializing the XML data to a stream or file.
- You can load XML data into a Document Object Model (DOM) tree from a stream or file. You can then manipulate the data as XML or as a DataSet. To do this, you must have an XML Schema to describe the structure of the data to the DataSet.
- You can create typed DataSets. A typed DataSet is a subclass of DataSet, with added properties and methods to expose the structure of the **DataSet**. To describe the XML representation of the DataSet, Microsoft Visual Studio® .NET generates an equivalent XML Schema definition for the typed DataSet.

Example

The following code loads the **Customer** table from the **Northwind Traders** database, saves it as XML to a temporary file, creates a new DataSet, and then loads the XML representation of the Customers table from the temporary file. The **System.IO** namespace is included for the **Path** class.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.IO; // included to get the temp path

namespace Samples {
    class DataSetXMLExample {
        static void Main(string[] args) {

            string tempfile = Path.GetTempFileName();

            string connectionString = @"data source=localhost;
Initial catalog=Northwind; integrated security=SSPI";
            string commandString = @"SELECT * FROM Customers";
            SqlDataAdapter dataAdapter = new SqlDataAdapter(
commandString, connectionString );

            DataSet myDataSet = new DataSet();
            dataAdapter.Fill( myDataSet, "Customers" );

            myDataSet.WriteXml( tempfile );
          Console.WriteLine( "Wrote Customer table to XML file
{0}", tempfile );

            DataSet data2 = new DataSet();
            data2.ReadXml( tempfile );

            int nRows = data2.Tables["Customers"].Rows.Count;
          Console.WriteLine( "Read {0} records from XML file
{1}", nRows, tempfile );
            // nRows is 91
        }
    }
}
```

This code sample is available on the Student Materials compact disc in the Samples\ModXB\XML folder.

Note For more information about how to fill a **DataSet** with an XML stream, see "DataSet class, XML" in the Visual Studio .NET online documentation.

For more information about obtaining data as XML from SQL Server, see "SQL Server .NET Data Provider, XML" in the Visual Studio .NET online documentation.

How to Change or Filter the View of the Data in the DataGrid

- Bind to DataViewManager to specify single or multiple column sort orders, including ascending and descending parameters
- ApplyDefaultSort to automatically create a sort order
- RowFilter to specify subsets of rows based on their column values
- RowStateFilter to specify which row versions to view
- DataGridTableStyle

Introduction

A **DataView** object provides a method for creating a customizable view for a single DataTable. You can sort and filter the view and specify what editing operations can be performed on the view's data.

You use a **DataViewManager** object to manage view settings for all the tables in a DataSet. The **DataViewManager** provides you with a convenient way to manage default view settings for each table. When you must bind a control to more than one table of a **DataSet**, binding to a **DataViewManager** is the ideal choice.

Sorting and filtering data using a DataView

The **DataView** provides several ways to sort and filter data in a **DataTable**, as shown in the following table.

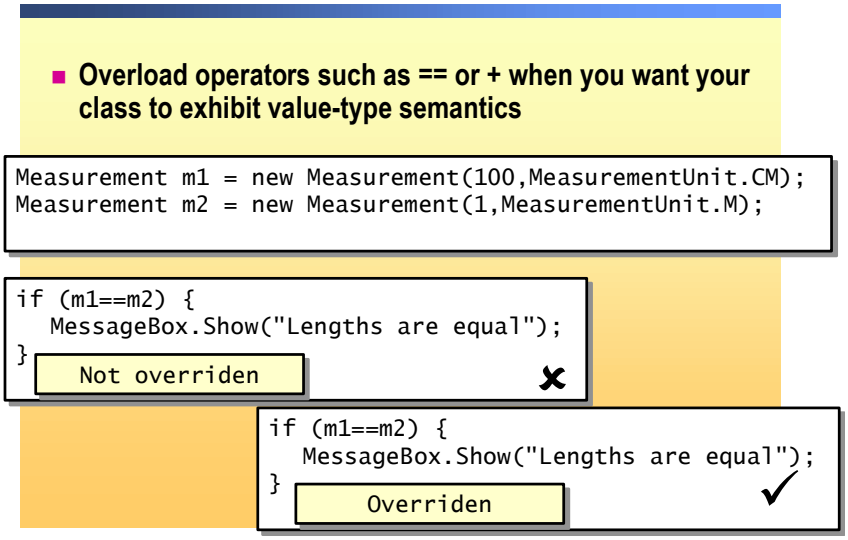
Use the:	To:
Sort property	Specify single or multiple column sort orders and include **ASC** (ascending) and **DESC** (descending) parameters
ApplyDefaultSort property	Automatically create a sort order, in ascending order, based on the primary key column or columns of the table
RowFilter property	Specify subsets of rows based on their column values
Find or **FindRows** methods of the **DataView**	Return the results of a particular query on the data rather than provide a dynamic view of a subset of the data
RowStateFilter property	Specify which row versions to view

Example

For example, the following code snippet sorts items in the **Customers** table by Country.

```
string connectionString = @"data source=localhost; Initial
catalog=Northwind; integrated security=SSPI";
string commandString = @"SELECT * FROM Customers";
dataAdapter = new SqlDataAdapter( commandString,
connectionString );

myDataSet = new DataSet();
dataAdapter.Fill( myDataSet, "Customers" );

DataViewManager dvm = new DataViewManager( myDataSet );

dvm.DataViewSettings["Customers"].Sort = "Country";

dataGrid1.SetDataBinding( dvm, "Customers");
```

How to Overload Operators



Introduction

You override operators such as == or + when you want your class to use value-type semantics. The following code demonstrates when you may want your class to display value-type semantics.

Overloading the == operator

```
public class Measurement {
  public decimal Length;
  public MeasurementUnit Unit;
  public Measurement(decimal len, MeasurementUnit t) {
      Length=len;
      Unit=t;
  }
}

public enum MeasurementUnit {
  M,
  CM,
  MM,
}
```

The preceding code contains a simple class, **Measurement**. Each instance of the class should store a **Length** and **Unit** value. Consider the following code:

```
Measurement m1 = new Measurement(100,MeasurementUnit.CM);
Measurement m2 = new Measurement(1,MeasurementUnit.M);
if (m1==m2) {
  MessageBox.Show("Measurements are the same");
}
else {
  MessageBox.Show("Measurements are not the same");
}
```

Although the two lengths are actually the same, the preceding test is testing for the instances referenced by the two measurement variables being the same, which in this case they are not.

For the preceding code to work, it is necessary to override the == operator. Overriding the == operator allows the class to behave more like a value type. In the following code, the class overrides the == operator:

```
public class Measurement {
  public decimal Length;
  public MeasurementUnit Unit;
  public Measurement(decimal len, MeasurementUnit t) {
      Length=len;
      Unit=t;
  }

  public static bool operator==(Measurement m1,
    Measurement m2) {
      decimal meters1 = 0;
      decimal meters2 = 0;
      switch (m1.Unit) {
          case MeasurementUnit.M:
              meters1 = m1.Length;
              break;
          case MeasurementUnit.CM:
              meters1 = m1.Length / 100;
              break;
          case MeasurementUnit.MM:
              meters1 = m1.Length / 1000;
              break;
      }
      switch (m2.Unit) {
          case MeasurementUnit.M:
              meters2 = m2.Length;
              break;
          case MeasurementUnit.CM:
              meters2 = m2.Length / 100;
              break;
          case MeasurementUnit.MM:
              meters2 = m2.Length / 1000;
              break;
      }
      if (meters1 == meters2) {
          return true;
      }
      else {
          return false;
      }
  }
}
```

The code in the overridden == operator converts each length to a length specified as a meter. The lengths can then be compared correctly. Notice how the overridden operator must be static; references to the two objects to be compared are passed as parameters.

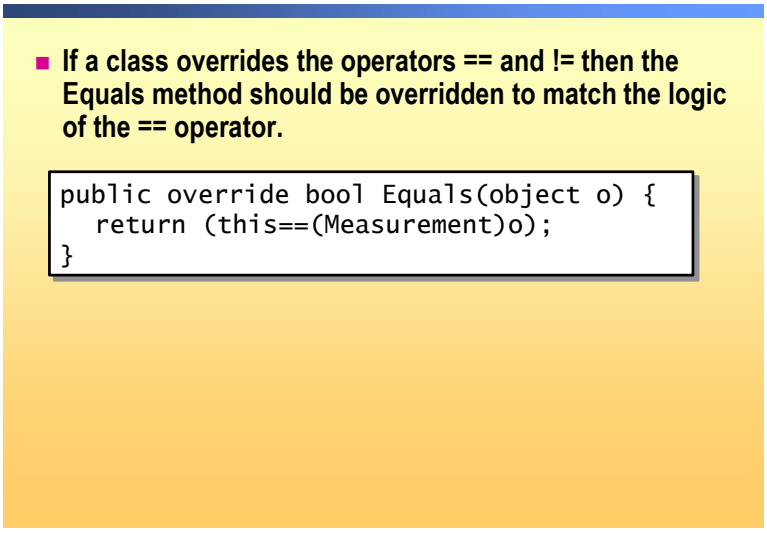
When overriding the == operator, it is necessary to also override the != operator. This is accomplished by adding the following code:

```
public static bool operator !=(Measurement m1,
      Measurement m2) {
          return !(m1 == m2);
}
```

Best practices

It is strongly recommended that you also override the **Equals** and **GetHashCode** methods when overriding the == and **!=** operators. Failure to override **Equals** and **GetHashCode** can generate confusion by those who develop with the class because the logic is not consistent.

How to Override and Implement Equals



Introduction

If your class must override the ==operator and != operator, you should also override the **Equals** method. You should use the == operator in the overridden **Equals** method to ensure that implemented logic is consistent between the == operator and **Equals** method.

Overriding the Equals method

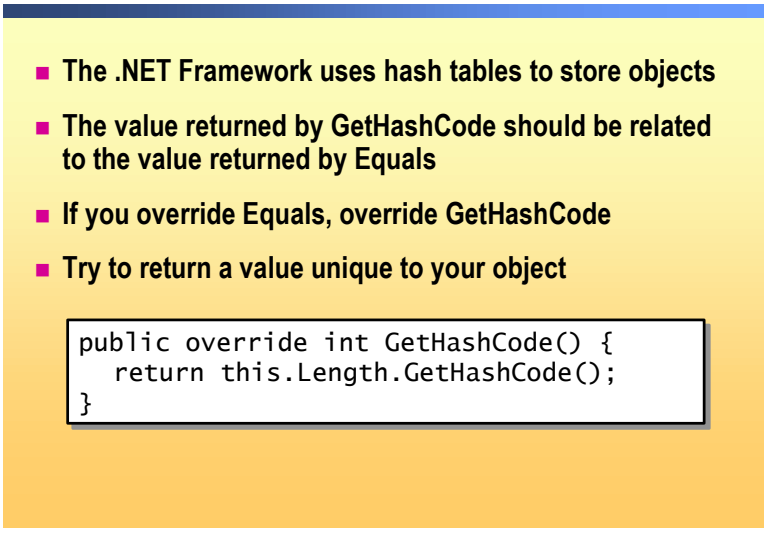
Expanding on the **Measurement** class example in the How to Overload Operators topic, it is good practice to override the **Equals** method, because the == operator and != operator are already overridden.

```
public class Measurement {
  public decimal Length;
  public MeasurementUnit Unit;
  public Measurement(decimal len, MeasurementUnit t) {
      Length=len;
      Unit=t;
  }

  public override bool Equals(object o) {
      return (this==(Measurement)o);
  }

  public static bool operator==(Measurement m1,
    Measurement m2) {
      decimal meters1=0;
      decimal meters2=0;
      switch (m1.Unit) {
          case MeasurementUnit.M:
              meters1=m1.Length;
              break;
          case MeasurementUnit.CM:
              meters1=m1.Length / 100;
              break;
          case MeasurementUnit.MM:
              meters1=m1.Length / 1000;
              break;
      }
      switch (m2.Unit) {
          case MeasurementUnit.M:
              meters2=m2.Length;
              break;
          case MeasurementUnit.CM:
              meters2=m2.Length / 100;
              break;
          case MeasurementUnit.MM:
              meters2=m2.Length / 1000;
              break;
      }
      if (meters1==meters2) {
          return true;
      }
      else {
          return false;
      }
  }

  public static bool operator !=(Measurement m1,
    Measurement m2) {
      return !(m1==m2);
  }
}
```

Best Practices

It is strongly recommended that you override **GetHashCode** methods when overriding the ==, **!=** and **Equals** operators.

How to Override GetHashCode



Introduction

You use hashing to efficiently find objects in lookup tables. A *hash table* is a method for speeding up a lookup process by producing a hash key for the objects in the table. You use a *hash key* to locate the area in the lookup table where you will most likely find the object, reducing the time spent looking for the object. For example, when executing a **switch** statement, C# uses the **Hashtable** class to quickly determine which branch to execute.

GetHashCode method

You can store any object in a hash table. To store an object, use the **GetHashCode** method to calculate the object's hash key.

Hash function properties

A hash function must have the following properties:

- If two objects of the same type represent the same value, the hash function must return the same constant value for either object.
- For the best performance, a hash function should generate a random distribution for all of the input.
- A hash function should be based on an immutable data member. *Immutable* means the data member, or string, and so on, does not change. The hash function should return exactly the same value regardless of any changes that are made to the object.

Caution Basing the hash function on a mutable data member can cause serious problems, including never being able to access that object in a hash table if the data member changes.

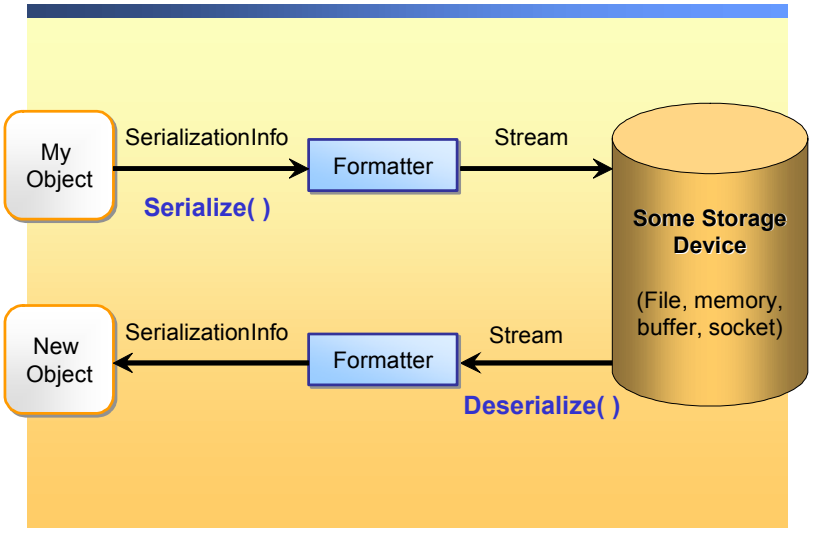
Using the Equals method with the GetHashCode method

If the **Equals** method determines that two objects are equal, the **GetHashCode** method must return the same integer for both objects. If a class overrides the **Equals** method, it should override the **GetHashCode** method.

For example, in the **Measurement** class example in the How to Override and Implement Equals topic and in the How to Overload Operators topic, to complete the implementation, you must override **GetHashCode**. This is accomplished by using the **GetHashCode** method of the data that is used in the == operator (Length).

```
public class Measurement {
  public decimal Length;
  public MeasurementUnit Unit;

  public Measurement(decimal len, MeasurementUnit t) {
      Length=len;
      Unit=t;
  }

  public override bool Equals(object o) {
      return (this==(Measurement)o);
  }

  public override int GetHashCode() {
      return this.Length.GetHashCode();
  }

  public static bool operator==(Measurement m1,
    Measurement m2) {
      decimal meters1=0;
      decimal meters2=0;
      switch (m1.Unit) {
          case MeasurementUnit.M:
              meters1=m1.Length;
              break;
          case MeasurementUnit.CM:
              meters1=m1.Length / 100;
              break;
          case MeasurementUnit.MM:
              meters1=m1.Length / 1000;
              break;
      }
      switch (m2.Unit) {
          case MeasurementUnit.M:
              meters2=m2.Length;
              break;
          case MeasurementUnit.CM:
              meters2=m2.Length / 100;
              break;
          case MeasurementUnit.MM:
              meters2=m2.Length / 1000;
              break;
      }
      if (meters1==meters2) {
          return true;
      }
      else {
          return false;
      }
  }

  public static bool operator !=(Measurement m1,
    Measurement m2) {
      return !(m1==m2);
  }
}
```

What Is Serialization?



Introduction

When you create an application, the ability to save information between user sessions is imperative. Occasionally, you will find it necessary to save the state of your component. You may do this to save personal information about users of your component, or to change the default configuration of a custom control.

Definitions and terms

Serialization is the term describing the process of converting the state of an object into a form (a linear sequence of bytes) that can be persisted or transported. The byte stream contains all of the necessary information to reconstruct or *deserialize* the state of the object for use later.

Example

When you serialize an object to a stream, you also serialize any additional object references that are required by the root object. After you save a set of objects to a stream, you can relocate the byte pattern as necessary.

For example, if you serialize a stream of objects to a memory stream, you can forward this stream to a remote computer or the Microsoft Windows® Clipboard, save it to a compact disc (CD), or simply store it in a file. It does not matter where the byte stream itself is stored. What matters is that this stream of 1s and 0s correctly represents the state of serialized objects.

.NET Framework serialization

The Microsoft .NET Framework provides two formatters for serialization: the **BinaryFormatter** class and **SoapFormatter** class. These classes convert an in-memory representation of your object to a stream of data.

- *Binary serialization* is accomplished by using **BinaryFormatter**, which converts the object graph to a binary stream, which is most useful for desktop applications.
- *XML serialization* is accomplished by using the **SoapFormatter**, which converts the object graph to SOAP format, which is most useful for Internet applications.

You can save these streams as files that you can deserialize and convert back to objects when needed.

Using serialization

Why would you want to use serialization? The two most important reasons are:

- To persist the state of an object to a storage medium so that you can re-create an exact copy at a later stage.

  It is often necessary to store the value of fields of an object to disk and then retrieve this data at a later stage. Although storing data and retrieving it later is easy to achieve without relying on serialization, this approach is often cumbersome and error prone, and becomes progressively more complex when you must track a hierarchy of objects.

  For example, if you write a large business application containing many thousands of objects and you must write code to save and restore the fields and properties to and from a disk for each object, serialization provides a convenient mechanism for achieving this objective with minimal effort.

- To send the object by value from one application domain to another.

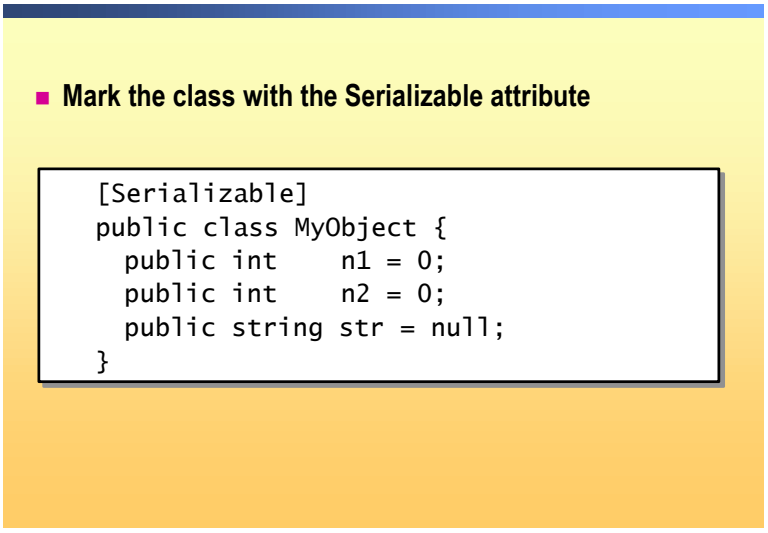
Reflection

The common language runtime manages how objects are laid out in memory and provides an automated serialization mechanism by using *reflection*. When an object is serialized, the name of the class, the assembly, and all of the data members of the class instance are written to storage. Objects often store references to other instances in member variables. When the class is serialized, the serialization engine keeps track of all referenced objects already serialized to ensure that the same object is not serialized more than once.

When the serialized class is *deserialized*, the class is re-created and the values of all the data members are automatically restored.

Tip For more about serialization guidelines, see the Visual Studio .NET documentation.

How to Use Binary Serialization



Introduction

The **BinaryFormatter** class defines two core methods that read and write an object to a stream:

- *Serialize ( )*. Serializes an object to a stream.
- *Deserialize ( )*. Deserializes a stream of bytes to an object.

The default configuration of **BinaryFormatter** also defines a number of properties that configure specific details regarding the serialization or deserialization process.

Example

Mark each class that you wish to persist to a stream with the **Serializable** attribute as follows:

```
[Serializable]
public class MyObject {
  public int n1 = 0;
  public int n2 = 0;
  public string  str = null;
}
```

The following code demonstrates how to serialize an instance of this class to a file:

```
MyObject obj = new MyObject( );
obj.n1 = 1;
obj.n2 = 24;
obj.str = "Some String";
IFormatter formatter = new BinaryFormatter( );
Stream stream = new FileStream("MyFile.bin", FileMode.Create,
   FileAccess.Write, FileShare.None);
formatter.Serialize(stream, obj);
stream.Close( );
```

This example uses a binary formatter to perform the serialization. All you must do is create an instance of the stream and the formatter that you intend to use, and then call the **Serialize** method on the formatter.

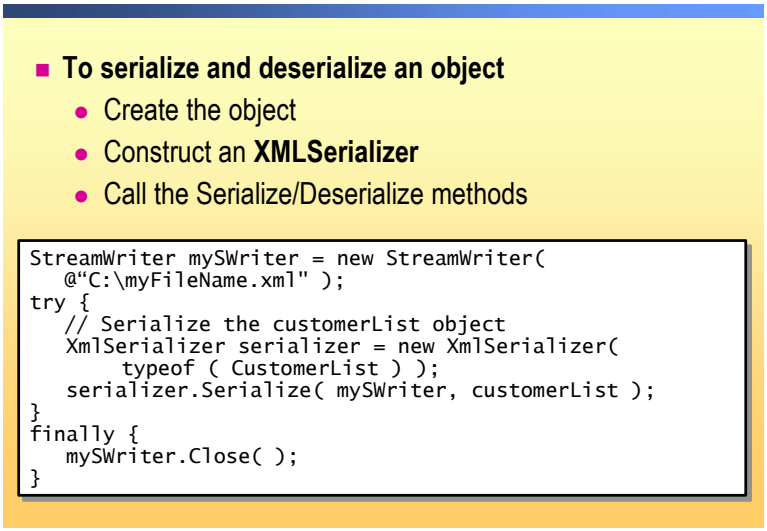
Restoring the object state

Restoring the object to its former state is just as simple. First, create a stream for reading and a formatter, and then instruct the formatter to deserialize the object. The following code snippet demonstrates all of the above:

```
IFormatter formatter = new BinaryFormatter( );
Stream stream = new FileStream("MyFile.bin", FileMode.Open,
  FileAccess.Read, FileShare.Read);
MyObject obj = (MyObject) formatter.Deserialize(fromStream);
stream.Close( );

Console.WriteLine("n1: {0}", obj.n1);
Console.WriteLine("n2: {0}", obj.n2);
Console.WriteLine("str: {0}", obj.str);
```

Tip All objects that are serialized with this formatter can also be deserialized with it.

How to Use XML Serialization



Introduction

XML serialization serializes only the public fields and property values of an object into an XML stream, and does not include type information. Because XML is an open standard, any application can process the XML stream, as needed, regardless of platform.

Note XML serialization does not convert methods, indexers, private fields, or read-only properties, except read-only collections. To serialize all of an object's fields and properties, both public and private, use the **BinaryFormatter** class instead of XML serialization.

XmlSerializer class

You can use the **XmlSerializer** class to serialize the following items.

- Public read/write properties and fields of public classes
- Classes that implement the **ICollection** interface or the **IEnumerable** interface.

  Note Only collections are serialized, not public properties.

- **XmlElement** objects
- **XmlNode** objects
- **DataSet** objects

SoapFormatter class

You can also use XML serialization to serialize objects into XML streams that conform to the SOAP specification. The other available formatter for serializing your types is the **SoapFormatter** class. The **SoapFormatter** class represents your object as a SOAP message, which is expressed in XML format. Use the **SoapFormatter** if portability is a requirement. Simply replace the formatter in the code that is used in the following example with **SoapFormatter**, and call the **Serialize** and **Deserialize** methods.

Serialize and Deserialize methods

To serialize an object, first create the object that is to be serialized and set its public properties and fields. To set the properties and fields, you must determine the transport format in which the XML stream is to be stored—either as a stream or as a file. For example, if the XML stream must be saved in a permanent form, create a **FileStream** object. When you deserialize an object, the transport format determines whether you will create a stream or file object. After you have determined the transport format, you can call the **Serialize** or **Deserialize** methods, as required.

Serializing an object

To serialize an object, complete the following tasks:

1. Create the object and set its public fields and properties.
2. Construct an **XmlSerializer** class by using the type of the object.
3. Call the **Serialize** method to generate either an XML stream or a file representation of the object's public properties and fields. The following example creates a file.

Example

```
MySerializableClass myObject = new MySerializableClass( );
XmlSerializer mySerializer = new
  XmlSerializer(typeof(MySerializableClass));
StreamWriter myWriter = new StreamWriter("myFileName.xml");
mySerializer.Serialize(myWriter, myObject);
```

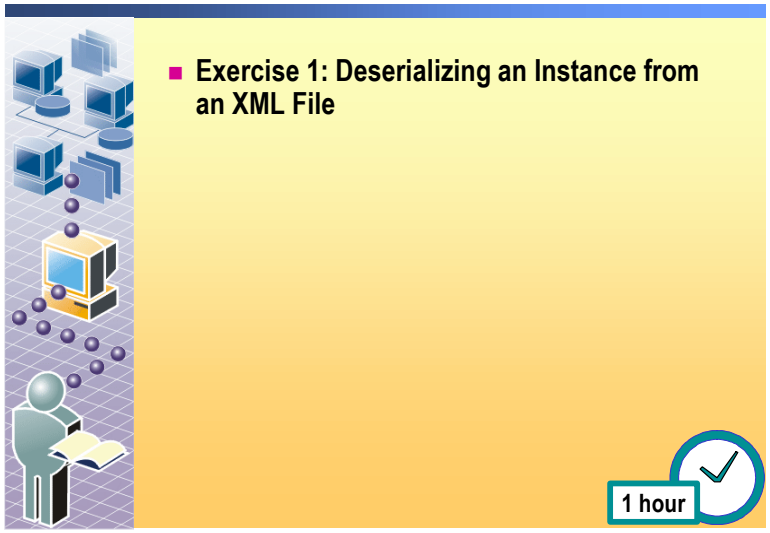
Deserializing an object

To deserialize an object, complete the following tasks:

1. Construct an **XmlSerializer** class by using the type of the object to deserialize.
2. Call the **Deserialize** method to produce a replica of the object. When deserializing, you must convert the returned object to the type of the original, as shown in the following example. The following example deserializes the object into a file; however, it could also be deserialized into a stream.

   ```
   MySerializableClass myObject;
   XmlSerializer mySerializer = new
   XmlSerializer(typeof(MySerializableClass));
   FileStream myFileStream = new FileStream("myFileName.xml",
      FileMode.Open);
   myObject = (MySerializableClass)
   mySerializer.Deserialize(myFileStream)
   ```

Lab B.1: Using Serialization



Objectives

After completing this lab, you will be able to:

- Serialize an object into an XML file.
- Deserialize an object from an XML file.

Prerequisites

Before working on this lab, you must have:

- Knowledge of the .NET Framework class library **XmlSerializer** class.
- Knowledge of the **Serializable** attribute.

Scenario

In this lab, you will add code to the Zoo Animal Information Display application that will add code to the **Zoo.Load** method that will deserialize the XML file and return an instance of the **Zoo** class.

The solution for this lab is provided in install_folder\Labfiles\LabXB_1\Solution_Code\Animals.sln. Start a new instance of Visual Studio .NET before opening the solution.

Estimated time to complete this lab: 60 minutes

Exercise 1
Deserializing an Instance from an XML File

In this exercise, you will complete the code in the **Zoo.Load** method to deserialize an XML file into an instance of the **Zoo** class.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open the *install_folder*\Labfiles \LabXB_1\animals.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* \Labfiles\LabXB_1, click **animals.sln**, and then click **Open**.
2. Locate the method **Zoo.Load** contained in the file zoo.cs.	**a.** In Solution Explorer, right-click **zoo.cs**, and then click **View Code**. **b.** Scroll through the code until you locate the method **public static Zoo Load(string filename);**.
3. To the **Zoo.Load** method, add code that creates a **FileStream** object that uses the filename passed to the method. Create an **XmlSerializer** object, and then use this object to deserialize the file stream.	▪ Refer to the content in this module for code samples of deserializing an object. You must modify this code to suit the application used in this lab. Note that the method returns a value of **True** if the method was successful.
4. Run the application. Open the file install_*folder***Labs\LabXB_1\AnimalData.xml**. Browse through the text that describes the animals.	**a.** On the standard toolbar, click **Start**. **b.** In the Zoo information window, click **File**, and then click **Open**. **c.** In the **Open** dialog box, click **AnimalData.xml**, and then click **Open**. At this point, your de-serialization code is being executed. If your code fails, attempt to debug your code and repeat this step. **d.** On the **View** menu, click **Next**.
5. Close the application. Save changes to your solution and then quit Visual Studio .NET.	**a.** Close the Zoo Information window. **b.** In Visual Studio .NET, on the **File** menu, click **Save All**. **c.** On the **File** menu, click **Exit**.