1. Introduction to the .NET Framework.

1.1. The .NET Framework and the Common Language Runtime

1.1.1. The .NET Framework is a foundation for software development and consists of:

a) The **common language run time**, which provides many of the core services required for program execution

b) The **.NET base class library**, which exposes a set of predeveloped classes to facilitate program development

The **Common Language Specification** defines a minimum set of standards that all languages using the .NET Framework must support.

The **Common Type System (CTS)** ensures type compatibility between components developed in different languages.

1.1.2. The **assembly** it the primary unit of a .NET application and consists of an assembly manifest which describes the assembly and one or more modules, which contain the source code for the application.

1.1.3. A .NET executable is stored as an **IL** file. When loaded, the assembly is checked against the security policy of the local system. If it is allowed to run, the first assembly is loaded into memory and JIT compiled into native binary code where it is stored for the remainder of the program's execution. The **JIT compiler** also compiles in advance and stores in memory execution branches that has not been executed.

1.2. The .NET Base Class Library

1.2.1. The .NET Framework base class library is a library of code that exposes functionality useful for application building. The base class library is organized into namespaces, which contain types and additional namespaces related to a common functionality.

1.2.2. Types can either be value types or reference types. A variable of a value type contains all of the data associated with that type. A variable of a reference type contains a pointer to an instance of an object of that type.

1.2.3. All of the data associated with a value type is allocated on the stack. When a variable of a value type goes out of scope, it is destroyed, and its memory is reclaimed. A variable of a reference type exists in two memory locations. The actual object data is allocated on the heap. A variable containing a pointer to that object is allocated on the stack. When that variable is called by a function, it returns the memory address for the object it refers to. When that variable goes out of scope, the reference to the object is destroyed, but the object itself is not. If any other references to that object exist, the object remains intact. If the object is left without any references to it, it is subject to garbage collection

1.2.4. Non-user-defined value types are created upon declaration and remain empty until they are assigned a value. Variables of a Reference type have a default value of null. Reference types must be instantiated after declaration to create the object. Declaration and instantiation can be combined into a single step without any loss of performance.

1.2.5. When a value type variable is assigned to another value type variable, the data contained within the first variable is copied into the second. When a reference type variable is assigned to another reference type variable, only the reference is copied, and both variables will refer to the same object.

1.2.6. You can use the *using* statement to allow reference to members of a namespace without using the fully qualified name. If you want to use an external library, you must create a reference to it.

1.2.7. When two types of the same name exist in more than one imported namespace, you must again use the fully qualified name in order to avoid a naming conflict. You can resolve namespace conflicts such as these by creating an alias.

```
using myAlias = MyNameSpaceTwo.Widget;
  // You can now refer to MyNameSpaceTwo as myAlias. The
  // following two lines produce the same result:
MyNameSpaceTwo.Widget anotherWidget= new myNameSpaceTwo.Widget;
MyAlias anotherWidget = new myAlias;
```

1.3. Using Classes and structures

1.3.1. User-defined types include classes and structures. Both can have members, which are fields, properties, methods, or events. Classes are reference types, and structures are value types.

1.3.2. You use the *class* keyword to create new classes. Structures are created by using the *struc* keyword. Both classes and structures can contain nested types.

1.3.3. User-defined types are instantiated in the same manner as predefined types except that both value types and reference types must use the *new* keyword upon instantiation.

1.3.4. Structures are best used for smaller, lightweight objects that contain relatively little instance data, or for objects that do not persist for long. Classes are best used for larger objects that contain more instance data and are expected to exist in memory for extended periods.

1.4. Using Methods

1.4.1. Methods perform the data manipulation that gives classes and structures their associated behavior. Methods can return a value, but do not have to. If a method doesn't return a value, it has a return type of *void*. Methods are called by placing the name of the method in the code along with any required parameters.

1.4.2. Methods can have parameters, which are values required by the method. Parameters are passed by value by default. You can pass parameters by reference with the *ref* keyword. For parameters of reference types, the behavior is the same whether passed by value or by reference. Visual C# allows you to specify output parameters from your method (*out* keyword).

1.4.3. The constructor is the first method called upon instantiation of a type. The constructor provides a way to set default values for data or perform other necessary functions before the object is available for use. Destructors are called just before an object is destroyed and can be used to run clean-up code. You cannot control when a destructor is called. A destructor in Visual C# is a method with the same name as the class preceded by a tilde (~).

1.5. Scope and access level

1.5.1. Access modifiers are used to control the scope of type members. There are five access modifiers, each provides varying levels of access:

- public
- internal <only for members of the assembly>
- private <only for themselves or within a containing type>
- protected < modifier to make a member defined in a base class available to an inherited class>

- protected internal

1.5.2. In classes and structures the default is public. In methods the default is private. In variables the default is private for classes and public for structures.

1.5.3. Nested types obey the same rules as non-nested types, but in practice can never have an access level greater than that of their parent type.

1.5.4. *static* members belong to the type but not to any instance of a type. They can be accessed without creating an instance of the type and are accessed using the type name instead of the instance name.

1.6. Garbage Collection

1.6.1. The .NET Framework provides automatic memory reclamation through the garbage collector. The garbage collector is a low priority thread that always runs in the background of the application. When memory is scarce, the priority of the garbage collector is elevated until sufficient resources are reclaimed.

1.6.2. You cannot be certain when an object will be garbage collected, so you should not rely on code in finalizers or destructors being run within any given time frame. If you have resources that need to be reclaimed as quickly as possible, provide a Dispose() method that gets called explicitly.

1.6.3. The garbage collector continuously traces the reference tree and disposes of objects containing circular references to one another in addition to disposing of unreferenced objects.

2. Creating the user interface

2.1. User interface design principles

2.1.1. Interface design is important because a visually consistent and logically designed interface will be easier to learn and will be more efficiently used.

2.1.2. Major elements of a user interface include forms, controls, and menus.

2.1.3. A good user interface will be carefully designed with attention to the following principles:

- Simplicity
- Positioning of Controls
- Consistency
- Aesthetics

2.1.4. Make choices that invite the target audience to use your application. Be aware of the cultural significance of your choices, and keep the principles of consistency and simplicity foremost in your mind when designing your interface.

2.2. Using Forms

2.2.1. Forms are the primary unit of the visual interface of a Windows Forms program. You should manage your forms in such a manner as to present a consistent, complete, and attractive visual interface to the end user. You can add forms to your application either at design time or run time, and you can use visual inheritance (using the inheritance picker) to create several forms with similar looks and layouts.

```
  //Displaying an already defined form called DialogForm
DialogForm myForm;
MyForm = new DialogForm;
  //Inheriting a form in code
```

```
public class myForm : MainForm
{
      //Class implementation here
}
```

2.2.2. To set the stratup form in a project, first define a Main method for the form and then set it up in the project properties page.

```
Static void Main()
{
      Application.Run(new myForm());
}
```

2.2.3. Forms have properties that control their appearance. Changing these properties will change the appearance of the form and sometimes controls hosted by the form. These properties include
BackColor , ForeColor , Text , Font , Cursor , BackGroundImage, Opacity

2.2.4. Forms have several intrinsic methods that you can use to control their lifetime and how they are displayed. Among them are

Form.Show	An instance of the form class os loaded in memory, visible and with the focus. For forms in memory it just make it visible.
Form.ShowDialog	Same that *Form.Show* but modal.
Form.Activate	For visible forms moves the form to the front an sets the focus.
Form.Hide	Same that setting the *visible* property to false.
Form.Close	Closes the form and removes it from memory

2.2.5. Each of these methods causes change in the visual interface and raises various events. Some of these events include

Load	Fired the first time a form is loaded in memory
Activated	Fired when a form recibes the focus (by Show, ShowDialog and Activate methods)
Deactivate	Fired when the form loses the focus (by Hide or Close –in active forms-) Used to validate input.
VisibleChanged	Fired when the Visible property changes. (by Show, ShowDialog, Hide and Close methods)
Closing	Raised when the form is in process of closing (by Close method or by clicking the close button). Used to validate that all the fields in the form have been filled out. To remaing the window open use: CancelEventArgsCancel = true;
Closed	Raised after a forms has been closed. (by the Close method or clicking the close button). Used to provide any clean-up code required for the form.

2.2.6. You can create specialized methods called event handlers that respond to these events. These methods are executed whenever the corresponding event is raised.

2.3. Using Controls and components.

2.3.1. You can set the tab order of the controls on your form by either setting the *TabIndex* property or by choosing Tab Order from the View menu and clicking on your controls to set the tab order.

2.3.2. Some controls can visually contain other controls, which can be used to create logical and visual subdivisions of your form. Examples of these controls include
Panel control
GroupBox control
TabPage within a TabControl

2.3.3. The *Dock* and *Anchor* properties allow you to implement automatic resizing or repositioning of the controls on your form. Setting the *Dock* property allows you to fix a control to an edge of your form. Setting the *Anchor* control allows you to specify whether your control remains fixed, floats, or changes size in response to the form resizing.

2.3.4. You can use the *controls* collection of a form to dynamically add controls at run time. To add a control, you must declare and create an instance of it, and add it to the *controls* collection of the appropriate form.

```
  //Get a control from the form myForm
```

```
Control aControl;
aControl = myForm.Controls[3];
  //Insert a label
Label aLabel = new Label();
aLabel.text = "Inserted label";
myForm.Controls.Add(aLabel);
  //Remove controls
myForm.Controls.Remove(Button      );
myForm.Controls.RemoveAt(3);  //An specific index
  //Add a control to a container control
Button aButton = new Button();
MyTabControl.TabPages[1].Controls.Add(aButton);
```

2.3.5. Additional controls can be added to the Toolbox by right-clicking the appropriate section of the Toolbox, and then selecting the appropriate control, or by browsing to the DLL that contains that control.

2.3.6. Extender provider components allow you to add design time properties to the controls on a form. At run time, these properties are commonly used to provide information to the user, such as in the form of a Tool Tip or Help. Those properties reside in the extender provider itself, not within the controls that they extend.

```
//Retrieve a tooltip of a button
string myToolTip;
myToolTip = toolTip1.GetToolTip(button1);
//Set the toolTip for a button:
toolTip1.SetToolTip(button1, "help message here")
```

2.4. Using Menus

2.4.1. Menus. The *MainMenu* control allows you create menus for your applications.To enhance menus use separator bars, access keys (Alt key and a letter), and shortcut keys (To enable instant access to menu commands).

2.4.2. Context menus are useful for enabling access to commands in a variety of contextual situations. Context menus can be created with the *ContextMenu* control and are created at run time in the same manner as main menus.

2.4.3. At run time you can enable and disable menu items, make menu items invisible, or display a check mark or radio button next to a menu item. You can also dynamically change the structure of menus by creating new menus with the *CloneMenu* method, merging multiple existing menus or adding new menu items to existing menus.

```
  //Disabling a menu item
menuItem1.Enabled =false;
  //Checking a menu item
menuItem1.Checked = true;
  //Making it invisible
menuItem1.Visible = false;
  //Cloning a menu (the File menu item into a context menu)
ContextMenu myContextMenu = new ContextMenu();
MyContextMenu.MenuItems.Add(fileMenuItem.CloneMenu());
MyButton.ContextMenu = myContextMenu;
  //Merging menus into a single menu
MainMenu1.MergeMenu(contextMenu1);
  //Adding a menu item with a method that handles the click event
MenuItem myItem;
myItem = new MenuItem("Item 1", new Eventhandler(myClick));
MainMenu1.MenuItems.Add(myItem);
  //Method that handles the click event
public void myClick (object sender, System.EventArgs e)
{
}
```

2.5. Validating user input

2.5.1. Form-level validation validates all fields on a form simultaneously.

2.5.2. Field-level validation validates each field as data is entered

2.5.3. The TextBox control contains several design time properties that restrict the values users can enter. These include

- MaxLength
- PasswordChar
- ReadOnly

- MultiLine The individual lines are stored as an array of strings in the *TextBox.Lines* collection and can be accessed by their index.

2.5.4. Keyboard events allow you to validate keystrokes; they are raised by the control that has the focus and is receiving input. The form will also raise these events if the *KeyPreview* property is set to true (It will raise before the controls that has the focus). These events are:

KeyDown / KeyUp	Mostly used to determine if Alt, Ctrl, or Shift were pressed through boolean properties in the *KeyEventArgs* parameter class. The *KeyCode* property gives the key pressed.
KeyPress	Raised only when an ASCII code (or Enter and BackSpace) is created. An instance of *KeyPressEventArgs* is passed as a parameter.

Examples:

```
private void textBox1_KeyUp(object sender, System.Windows.Forms.KeyEventArgs e)
{
    if (e.Alt == true)   MessageBox.Show("The ALT key is still down");
}

private void textBox1_KeyDown(object sender, System.Windows.Forms.KeyEventArgs e)
{
MessageBox.Show(e.KeyCode.ToString());
}
```

2.5.5. The *Char* structure contains several static methods that are useful for validating character input. These include

Char.IsDigit
Char.IsLetter
Char.IsLetterOrDigit
Char.IsPunctuation
Char.IsLower
Char.IsUpper

```
private void textBox1_KeyPress (object sender,
                                System.Windows.Forms.KeyPressEventArgs e)
{   if (Char.IsDigit(e.KeyChar) == true)
      MessageBox.Show("You pressed a number key");
}
```

2.5.6. Altough any control implement the *Focus* method, disabled or invisible controls cannot receive the focus:

```
//Check if textBox1 can receive the focus
if (textBox1.CanFocus == true)
      textBox1.Focus();
```

2.5.7. Focus events occur in the following order:

Enter	The focus arrives
GotFocus	First obtain the focus
Leave	The focus leaves
Validating	Before the control loses focus
Validated	After the controls has successfully been validated
LostFocus	The focus was lost

2.5.8. The *Validating* event occurs before the control loses focus and should be used to validate user input. This event occurs only when the *CausesValidation* property of the control that is about to receive the focus is true. To keep the focus from moving away from the control in the Validating event handler, set the *CancelEventArgs.Cancel* property to true in the event handler.

```
private void textBox1_Validating(object sender,
                                 System.ComponentModel.CancelEventArgs e)
{
```

```
   // Checks the value of textBox1
if (textBox1.Text == "")
  // Resets the focus if there is no entry in TextBox1
      e.Cancel = true;
}
```

2.5.9. Form Level Validation validates all the controls of the form at once.. Example:

```
private void btnValidate_Click(object sender, System.EventArgs e)
{
  // Loops through each control on the form
foreach (System.Windows.Forms.Control aControl in this.Controls)
    {
       // Checks to see if the control being considered is a
       // Textbox and if it contains an empty string
    if (aControl is System.Windows.Forms.TextBox & aControl.Text == "")
       {
          // If a textbox is found to contain an empty string, it is
          // given the focus and the method is exited.
         aControl.Focus();
         return;
       }
     }
}
```

2.5.10. The ErrorProvider component allows you to set an error for a control at run time that displays a visual cue and an informative message to the user. You can also set it up at design time (by setting the control property "Error on <ErrorProviderName>") but the error will be displayed aways. To display an error at run time, use the *ErrorProvider.SetError* method. Make sure to set *CausesValidation* property for the control to true (the default).

```
private void pswordTextBox_Validating(object sender,
        System.ComponentModel.CancelEventArgs e)
{
// Validate the entry
if (pswordTextBox.Text == "")
// Set the error for an invalid entry      myErrorProvider.SetError
(pswordTextBox,
 "Password cannot be blank!");
else
// Clear the error for a valid entry-no error will be
// displayed
myErrorProvider.SetError(pswordTextBox, "");
}
```

3. Types and Methods

3.1. The .NET Framework provides a robust, strongly typed data system (object of on type cannot be freely echanged with objects of a different type). Different types represent integer numbers, floating-point numbers, Boolean values, characters, and strings.

3.1.1. Integer can receive hexadecimal assignments using 0x:

```
Int myInteger;
MyInteger = 0x32EF
```

3.1.2. The *System.Object* type is the supertype of all types in the .NET Framework.

```
Object myObject;
myObject = 543;
myObject = new System.Windows.Forms.Form();
```

3.1.3. All the datatypes support four methods:

- *Equals*. Determines whether two instances are equal
- *GetHashCode*. Serves as a hash function for a particular type
- *GetType*. Returns the type object for the current instance

- *ToString*. Returns a human-readable form of the object

3.1.4. Conversion between types can take one of two forms:
Implicit conversions, which are performed automatically by the run time. The only conversions that can be performed implicitly are conversions where there is no possible loss of data, such as a conversion from a lower precision type to a higher precision type.
Explicit conversions are performed explicitly in code. Conversions of this type can potentially lead to the loss of data and should be undertaken with caution. .

3.1.5. Boxing allows you to treat a value type like a reference type. Unboxing converts a boxed reference type back to a value type.

3.1.6. The .NET types provide built-in functionality specific to the type. The *Parse* method is implemented by all of the built-in value types and is useful for converting strings to value types. The *String* class exposes several useful functions that allow you to easily manipulate strings.

```
int I;string S;
S = "1234";
// I = 1234
I = int.Parse(S);
```

3.2. Constans, Enums, Arrays, and Collections

3.2.1. Constants and enumerations make your code easier to read and maintain, and less error prone, by substituting friendly names for frequently used values. You define a constant with the *const* keyword. The *enum* keyword is used to declare an enumeration and they are statics members of its class. Constants can be of any data type. Enumerations must be of a numeric integral type.

```
//Declare a constant
public const double Pi = 3.14159265;
//Declare an enumerator, the default assigments start from 0.
public enum DaysOfWeek
{
  Monday = 1,
  Tuesday = 2,
  Wednesday = 3,
  Thursday = 4,
  Friday = 5,
  Saturday = 6,
  Sunday = 7
}
//Define an enum using a numeric integral type
public enum DaysOfWeek : byte
       // additional code omitted
// This uses the Numbers enum from the previous example
MessageBox.Show(((int)Numbers.two * 2).ToString()); // Displays 4
//Using enums as parameters
public void ScheduleDayOff(DaysOfWeek day)
{
   switch(day)
   {
     case DaysOfWeek.Monday:
     // Implementation code omitted
     break;
     case DaysOfWeek.Tuesday:
     // Implementation code omitted
     break;
     // Additional cases omitted
   }
}
```

3.2.2. Arrays can be one dimensional or multidimensional. Multidimensional arrays can be either rectangular or jagged. In rectangular arrays, every member of each dimension is extended into the other dimensions by the same length. In jagged arrays, individual members of one dimension can be extended into other dimensions by different lengths. In either case, the more dimensions, the more complex the array.

```
  // Declare and initialize an array of 32 integers (0 -31)
int[] myIntegers = new int[32];
  // Declare the array (no dimension)
int[] myIntegers;
  // Initialize the array with 32 members (set dimension)
myIntegers = new int[32];
  // This line reinitialized the array
myIntegers = new int[45];
```

```
  // Managing an Array of reference types
objectWidget[] Widgets = new Widget[11]; //Null references
  // Assigns Widgets[0] to a new Widget
Widgets[0] = new Widget();
  // Assigns Widgets[1] to an existing Widget object
Widget aWidget = new Widget();
Widgets[1] = aWidget;
  // Loops through Widgets and assigns 2 through 10 to a new object
for (int Counter = 2; Counter < 11; Counter++)
   {
    Widgets[Counter] = new Widget();
   }
```

```
  // Rectangular arrays (same number of rows and columns
int[ , ] intArrays = new int[5, 3];
  // Declares a two-dimensional array (2 rows,3 columns)
int[ , ] intArrays = {{1, 2, 3}, {4, 5, 6}};
 intArrays[0,0] = 1; intArrays[0,1] = 2; intArrays[0,2] = 3;
 intArrays[1,0] = 4; intArrays[1,1] = 5; intArrays[1,2] = 6
  // Declares a cubical array and sets initial values
int[ , , ] cubeArray = {{{7, 2}, {1, 4}}, {{3, 5}, {7, 9}}};
 cubeArray[0,0,0] = 7
 cubeArray[0,0,1] = 2
 cubeArray[0,1,0] = 1
 cubeArray[0,1,1] = 4
 cubeArray[1,0,0] = 3
 cubeArray[1,0,1] = 5
 cubeArray[1,1,0] = 7
 cubeArray[1,1,1] = 9
  // Declares a complex array of 3 x 3 x 4 x 5 x 6 members
int[ , , , , ] complexArray = new int[3, 3, 4, 5, 6];
```

```
  //Jaged Arrays (Each row can have different columns)
  // Declares an array of 3 arrays
string[][] Families = new string[3][];
  // Initializes the first array to 4 members and sets values
Families[0] = new string[] {"Smith", "Mom", "Dad", "Uncle Phil"};
  // Initializes the second array to 5 members and sets values
Families[1] = new string[] {"Jones", "Mom", "Dad", "Suzie","Little  Bobby"};
  // Initializes the third array to 3 members and sets values
Families[2] = new string[] {"Williams", "Earl", "Bob"};
```

3.2.3. Collections allow you to manage groups of objects, which can be of the same type or different types. There are several types of collections, and all are available in the *System.Collections* namespace. The *System.Collections.ArrayList* provides the basic functionality suitable for most applications.

```
 Widget myWidget = new Widget();
  System.Collections.ArrayList myList = new System.Collections.ArrayList();
// Adds the Widget to the collection
  myList.Add(myWidget);
// Removes the Widget from the collection
  myList.Remove(myWidget);
```

Removing an unexistent object is just ignored.

3.2.4. You can use the *foreach* statement to iterate through the members of an array or collection, but you cannot alter members using this statement. To loop through an array or collection and alter the members, use the *for*.

```
  int[] myArray = new int[] {1,2,3,4,5};
  foreach (int I in myArray)
  {   MessageBox.Show(I.ToString());}

// Assumes that myList is an ArrayList that contains Strings and other types
  foreach (object o in myList)
  {
   if (o.GetType() == typeof(string))
     { MessageBox.Show(o.ToString()); }
  }
```

3.3. Implementing Properties

3.3.1. A property is essentially a specialized method that looks like a field. Properties allow you to expose member variables or objects and provide code to validate property values or perform other functions when the property is

accessed. Read-only and write-only properties are useful for limiting the ability of the client to read or write to your properties.

```
// Creates a private local variable to store the value
   private string mText;
// Implements the property
  public string MyText{
    get
     {
      return mText;
     }
    set
     {
      mText = value;
     }
  }
//Readonly property, the value of the variable is set in the constructor
  private readonly int mInt;
  public int InstanceNumber
  {
    get
     {
      return mInt;
     }
   }
```

3.3.2. A default property allows you to access values exposed by a property without using the property name for qualification. Default properties must be parameterized properties. The Visual C# equivalent of a default property is an indexer.

```
//In indexers the name of the object itself is used to access the property
// The name of the property is set to "this"
  // Creates an array to store the values exposed by the indexer
private int[] IntArray;
  // The index variable in brackets is used to retrieve the correct item
public int this[int index]
  {
    get
      {
       return IntArray[index];
      }
    set
      {
       IntArray[index] = value;
      }
  }
```

3.3.3. You can use collection properties to expose groups of objects exposed by your classes. There are several ways to do this, each with its own advantages and drawbacks:

3.3.3.1. Create a simple collection property that returns a private member collection object. This approach is the least robust.

```
//Implement a read-only property that returns the collection
private readonly System.Collections.ArrayList mWidgets = new
                                          System.Collections.ArrayList();
public System.Collections.ArrayList Widgets
{
   get
   {
      return mWidgets;
   }
}
```

3.3.3.2. Create a property that returns an object and accesses a private member collection object by index. This approach is more robust, but presents some limitations in the usability of your class.

```
//create a pair of methods that get and set the object at the
appropriate
```

```
//index in a private collection. In the get method, explicitly convert
//the object returned by the collection to the appropriate type.

private System.Collections.ArrayList mWidgets = new
                                              System.Collections.ArrayList();
public Widget GetWidget(int I)
{
   return (Widget)mWidgets[I];
}
public void SetWidget(int I, Widget Wid)
{
   mWidgets[I] = Wid;
}
```

3.3.4. Implement a strongly typed collection class. This approach is most developmentally intensive, but provides the most robust method for exposing a strongly typed collection of classes.

```
//Implemententing strong type collections
//Class to be used as the datatype for the collection
class Widget
{
}


public class MyCollection : System.Collections.CollectionBase
{
  public MyCollection()
        {
                // TODO: Add constructor logic here
        }
  public void Add(Widget aMember)
        {
                List.Add(aMembert);
        }

  public Widget this[int index]
        {
                get
                {
                        return (Widget) List[index];
                }
        }
  public void Remove(Widget aMember)
        {
                if (List.Contains(aMemeber) == true)
                {
                        List.Remove(aMember);
                }
        }
}
  //instantiate the collection
myCollection TestCollection = new MyCollection();
  //Create an object and add it to the collection
Widget TestClass = new Widget();
TestCollection.Add(TestClass);
  //Retrieve a member from the collection
Widget TestClass2 = TestCollection[0];
  //Two ways of removing members in the collection
```

```
TestCollection.Remove(TestClass);
TestCollection.RemoteAt(0);
```

3.4. Implementing Delegates and Events.

3.4.1. Events are members of classes that are used to communicate interesting occurrences between classes. An instance of a class can raise a member event to send out the message. The event can be handled by methods that are designated as event handlers. These methods execute when the event is raised.

3.4.2. Delegates provide the functionality behind events. A delegate is a strongly typed function pointer. It can be used to invoke a method without making an explicit call to that method. They are static members of its class. The creation of associations between events and event handlers requires the use of delegates.

```
//Declare a delegate with the signature of the method that it can call
  public delegate int myDelegate(double D);
// This method is the target for the delegate
  public int ReturnInt(double D)
    {
       // Method implementation omitted
    }
// This line creates an instance of the myDelegate delegate that
// specifies ReturnInt
  public void amethod()
   {
     myDelegate aDelegate = new myDelegate(ReturnInt);
   //Once declared and assigned, you can use the delegate to invoke the
   //method in the same manner in which you would make a function call
     aDelegate(12345);
   }
```

3.4.3. Events are declared as members of the class. The event must reference an appropriate delegate. A member event is raised by calling the event by name. Events can return a value.

```
//Declaring and raising events
//Declare the event and supply the delegate that the even will use
//It can be Public, Private or protected
  public delegate void calculationDelegate(double d);
  public event calculationDelegate CalculationComplete;
//Raise the event like calling a method
  CalculationComplete(66532);
```

3.4.4. Methods that handle events are called event handlers. An event handler is a method that is called through a delegate when an event is raised, If you raise an event that has no event handlers in Visual C#, an error will result.You can create associations between events and event handlers in code. You create the event handler by using the += operator to associate a delegate with the event.

```
// These examples assume the existence of a method called
// DisplayResults that has the appropriate signature for
// CalculationDelegate
//Creates a new delegate to create the association, delegates are static
//members of the class where they are defined
//(assume here that Account is an instance of AccClass
  Account.CalculationComplete +=
                       new  AccClass .calculationDelegate(DisplayResults);
// Creates an association with an existing delegate
  CalculationDelegate calc =
                       new AccClass.calculationDelegate(DisplayResults);
  Account.CalculationComplete += calc;
```

3.4.5. Default delegates are provided for events of the controls and classes of the .NET framework, for example for controls in *System.Windows.Forms* there is *System.EventHandler*:

```
button1.Click += new System.EventHandler(clickHandler);
```

3.4.6. Events can have multiple event handlers associated with them, and multiple events can be handled by the same method. Event handlers can also be removed dynamically.

```
// Assumes the existence of a method called ClickHandler with the
```

```
// correct signature to receive button click events
  button1.Click += new System.EventHandler(ClickHandler);
  button2.Click += new System.EventHandler(ClickHandler);
```

4. Object-Oriented programming and Polymorphism
   4.1. Introduction to Object-Oriented Programming
      4.1.1. Objects are composed of members.
         4.1.1.1. Members are properties, fields, methods, and events, and they represent the data and functionality that comprise the object.
            4.1.1.1.1. Fields and properties represent data members of an object.
            4.1.1.1.2. Methods are actions the object can perform.
            4.1.1.1.3. Events are notifications an object receives from or sends to other objects when something interesting or noteworthy has happened.
      4.1.2. The representation of real-world objects as programmatic constructs is called abstraction. Programmatic objects can represent real-world objects through their implementation of members.
      4.1.3. Classes are the blueprints for objects. When an object is created, a copy of the class is created in memory, and values for member variables are initialized. A class can act as a template for any number of distinct objects.
      4.1.4. Encapsulation is a principle of object-oriented programming. An object should contain all of the data it requires and all of the code necessary to manipulate that data. The data of an object should never be made available to other objects. Only properties and methods should be exposed in the interface.
      4.1.5. Polymorphism is the ability of different objects to expose different implementations of the same public interface. Two major types of polymorphism:
         4.1.5.1. Interface polymorphism. An interface defines a contract for behavior. It specifies what members must be implemented but provides no details as to their implementation. An object can implement many unrelated interfaces, and many diverse objects can implement the same interface.
         4.1.5.2. Inheritance polymorphism. Objects can inherit functionality from one another. An inherited class retains the full implementation of its base class, and instances of inherited classes can be treated as instances of the base class. Inherited classes can implement additional functionality, as required.
   4.2. Overloading Members
      4.2.1. Overloading allows you to create multiple methods with the same name but different implementations. Overloaded methods must differ in signature, but can have the same or different return types and access levels. You declare overloaded methods just as you would declare a regular method.
      4.2.2. C# allows you to define custom behaviors for operators when used with user-defined types. Overloaded operators must be public and static. You use the *operator* keyword to declare an overloaded operator. The syntax is:
      public static *type* operator op (Argument1[, Argument2])
      { implementation}
      At least one argument must be of type *type*
      Examples:

```
  public struct HoursWorked
{
   float RegularHours;
   float OvertimeHours;
   // This is the overloaded operator defined in the previous example
   public static HoursWorked operator + (HoursWorked a, HoursWorked b)
   {
      HoursWorked Result = new HoursWorked();
      Result.RegularHours = a.RegularHours + b.RegularHours;
      Result.OvertimeHours = a.OvertimeHours + b.OvertimeHours;
      return Result;
   }
   // An additional implementation of the + operator is defined below.
   public static HoursWorked operator + (HoursWorked a, int b)
   {
      HoursWorked Result = new HoursWorked();
      Result.RegularHours = a.RegularHours + b;
      return Result;
   }
}
```

```
  // This example assumes that the variables Sunday and Monday represent
  // instances of the HoursWorked struct that have been created and
  // had appropriate values set.

HoursWorked total = new HoursWorked();
total = Sunday + Monday;
```

4.3. Interface Polymorphism

4.3.1. An interface defines a contract for behavior. It defines the members that will be exposed through the interface, and the parameters and return types of those members. Any object that implements an interface can interact with any object that requires that interface. Classes and structures can both implement interfaces, and each can implement multiple interfaces.

4.3.2. The access modifier of the interace (Public, private etc), determines the access of its members. Interfaces can define methods, events and properties but no fields.

```
public interface IDrivable
{
   //Methods
   void GoForward(int Speed);
   void Halt();
   int DistanceTraveled();
  //Properties
  int FuelLevel
   {
      get;
      Set;
   }
  //Events, you must designate the delegate for the event
  event System.EventHandler OutOfFuel;
}
```

4.3.3. Casting to an interface can be done as follows:

```
//Any variable that implements this interface can be passed as a paramenter.
public void GoSomewhere(IDrivable v)
{
   // Implementation omitted
}
//Here truck implements the interface and can be casted:
Truck myTruck = new Truck();
IDrivable myVehicle;
// Casts myTruck to the IDrivable interface
myVehicle = (IDrivable)myTruck;
```

4.3.4. In C#, you can implement interface members in two ways:

4.3.4.1. By implementing a member with the same name, signature, and access level as the member defined in the interface. This member will be available to both the class that implements it and to the interface.

```
public interface IDrivable
{
   void GoForward(int Speed);
}
public class Truck : IDrivable
{
   public void GoForward(int Speed)
   {
      // Implementation omitted
   }
}
```

4.3.4.2. By explicitly implementing the interface member by using the fully qualified name of the member. If implemented in this manner, the member will be available only to the interface (casting). No access modifier is defined because utilizes the same from the interface.

```
public class Truck : Idrivable
```

```
{
   void IDrivable.GoForward(int Speed)
   {
      // Implementation omitted
   }
}
```

4.4. Inheritance Polymorphism

4.4.1. You can create classes that combine all of the functionality of a previously defined class with new specialized functionality through inheritance. Inherited classes contain all of the members of their base class, and instances of an inherited class can polymorphically act as instances of their base class.

4.4.2. To make a class non inheritable use the keywork *sealed*

```
public sealed class Aclass
{
   // Implementation omitted
}
```

4.4.3. You can add additional members to your inherited class to provide custom functionality. You can also provide new implementation for existing inherited members. There are two ways to do this:

4.4.3.1. Overriding members. Using the *Override* keyword in a method with the same name, signature, return type and access level. In the base class the method has to have the *virtual* keyword. The inherited method is replaced, even if the class (child) is casted to its base class type (parent):

```
public class SportsCar : Car
{
   public override void GoForward(int Speed)
   {
      // implementation omitted
   }
}
```

4.4.3.2. Hiding members. Replaces a base class member with a new implementation. The new method can change only access level and return type. Use the *new* keyword in the method that is hiding another method:

```
public class MyBaseClass
{
   public string MyMethod(int I) {  }
}
// This class inherits the base class
public class MyInheritedClass : MyBaseClass
{
   internal new int MyMethod(int I)  {   }
}
```

4.4.3.2.1. Hidden members are not inheritable. If a class that has a hidden member is inherited, the hidden member is not inherited, and the new class will expose the base member.

4.4.3.2.2. The type of the variable determines whether a hidden member or the original member is called.

```
MyBaseClass theBase;
MyInheritedClass theInherited = new MyInheritedClass();
// Both refer to the same object, but they have different types.
theBase = theInherited;
// The new member will be called
theInherited.MyMethod(42);
// The original implementation of the member will be called.
theBase.MyMethod(42);
```

4.4.4. You can always access the base class implementation of a member by using the *base* keyword. You can use the *protected* access modifier to make a member defined in a base class available to an inherited class.

```
public override void MyMethod()
{
   base.MyMethod();
   // Additional implementation omitted
}
```

4.4.5. Abstract classes allow you to create a class that defines the interface of the class but provides part or none of the implementation. Abstract classes cannot be instantiated on their own; rather, they must be inherited. To provide an implementation for an abstract member in an inherited member, you must override the member. An abstract class can be inherited from another abstract class and optionally can implement the abstract members from the base class.

```
public abstract class Car
{ //Members to be implemented by inheriting classes
   public abstract void GoForward(int I);
   public abstract int CheckSpeed();
   public abstract string Color
   {
      get;
      set;
   }
}
public class MyCar : Car
{
   public override void GoForward(int I)
   {
      // Specific implementation of this method would go here
   }
   public override int CheckSpeed()
   {
      // Implementation for this method goes here
   }
   public override string Color
   {
      get
      {
         // Both the getter and the setter must be implemented
      }
      set
      {
         // Setter implementation goes here
      }
   }
}
```

5. Testing and debuging your application
   5.1. Testing and debugging your application
      5.1.1. There are three types of errors you might encounter while writing your program:
         - Syntax errors, which represent syntax the compiler cannot interpret
         - Run-time errors, which occur when an impossible operation is attempted
         - Logical errors, which compile and execute correctly, but return unexpected results
      5.1.2. You can use Break mode to examine your code line-by-line to identify and correct errors. Visual Studio .NET provides several options for stepping through code:
         - Step Into allows you to step through each line of code as it executes, including calls to other functions.
         - Step Over allows you to step through each line of code as it executes, but steps over calls to other functions.
         - Step Out executes the remainder of code in the current function and stops at the next line in the function that called it, if applicable.
         - Run To Cursor allows you designate a line with the cursor and execute all the code leading up to that line.
         - Set Next Statement allows you to set the next statement to be executed, skipping and not executing any intermediate lines.
      5.1.3. Breakpoints are designated lines of code where execution halts and the application enters Break mode when debugging. Breakpoints can have conditions attached that determine whether the application breaks or not. You can use the Breakpoints window to manage, create, disable, or clear breakpoints.
      5.1.4. Visual Studio .NET provides several tools that can be used to evaluate your program. The Locals, Autos, and Watch windows allow you to observe program variables while the application is running. The Command window allows you to execute code and print variable and property values to the window. Additional windows allow you to observe a variety of data regarding your application.
   5.2. Using the Debug and Trace Classes
      5.2.1. The *Debug* and *Trace* classes allow you to display and log messages about application conditions while the program is executing. The *Debug* and *Trace* classes are identical. Both classes expose methods that are used to create trace output. *Debug* is used when developing and *Trace* when the program has been released.
      5.2.2. *Debug* and *Trace* have 6 static methods to write output:

```
  // Writes text to the Listeners collection.
Trace.Write("Trace Message 1");
  // Writes text and a carriage return to the Listeners collection.
```

```
Trace.WriteLine("Trace Message 2");
  // Writes text if the supplied expression is true
Debug.WriteIf(X==Y, "X equals Y");
  // Writes text and a carriage return if the supplied expression is true
Debug.WriteLineIf(X==Y, "X equals Y");
  // Writes output and displays a message box if the condition is false
Trace.Assert(X==Y, "X does not equal Y!");
  // Writes output and displays a message box unconditionally
Debug.Fail("Drive B is no longer valid.");
```

5.2.3. Trace messages can be indented by using the *Indent* and *UnIndent* methods

```
// Increases the IndentLevel by one
Trace.Indent();
```

5.2.4. Output from Trace and Debug statements is received by members of the *Trace.Listeners collection*, a collection of objects that are able to receive Trace output and process it. There are three kinds of Trace Listeners:

5.2.4.1. *DefaultTraceListener* . Used by Visual Studio.

5.2.4.2. *TextWriterTraceListener*. writes its output as text, either to a Stream object or to a TextWriter object.

```
  // This line opens the specified file or creates it if it does not exist.
System.IO.FileStream myLog = new System.IO.FileStream("C:\\myFile.txt",
                                        System.IO.FileMode.OpenOrCreate);
  // Creates the new TraceListener that specifies myLog as the target for output
TextWriterTraceListener myListener = new TextWriterTraceListener(myLog);
  // Adds myListener to the Listeners collection
Trace.Listeners.Add(myListener);
```

```
  //You have to flush after making a write
Trace.Flush
  //Or you can set autoflush to do it after every write
Trace.AutoFlush = true;
```

5.2.4.3. *EventLogTraceListener* . Sends the output to an event log

```
//Declare an instance of an EventLog object and assign it either an
//existing event log or a new event log.
  EventLog myLog = new EventLog("Debug Log"); //New log created
//Set the Source property for the EventLog.
//If the Source property is not set, an error will result.
  myLog.Source = "Trace Output";
//Create a new instance of EventLogTraceWriter that specifies the
//new event log as the target for Trace output.
  EventLogTraceListener myListener = new EventLogTraceListener(myLog);
//If necessary, set the Trace.AutoFlush property to true,
//or call Trace.Flush after each write.
```

5.2.5. Trace switches are used to configure Trace statements. BooleanSwitch objects are either on or off. *TraceSwitch* objects expose a *TraceSwitch.Level* property that is used to determine the verbosity of Trace statements.

```
//Contructor takes a displayName and a description
BooleanSwitch myBooleanSwitch = new BooleanSwitch("Switch1",
   "Controls Data Tracing");
TraceSwitch myTraceSwitch = new TraceSwitch("Switch2",
   "Controls Form1 Tracing");
```

5.2.6. The *TraceSwitch.Level* property determines the level of detail to be shown, and it takes a *TraceLevel* enum with any of the following values:

- *TraceLevel.Off* - Represents a TraceSwitch that is not currently active. The integer value is 0.
- *TraceLevel.Error* - only represents very brief error messages. The integer value is 1.
- *TraceLevel.Warning* - Represents error messages and warnings. The integer value is 2.
- *TraceLevel.Info* - Represents error messages, warnings, and short informative messages. The integer value is 3.
- *TraceLevel.Verbose* - Represents error messages, warnings, and detailed descriptions of program execution. The integer value is 4.

5.2.7. Additionally, the *TraceSwitch* class exposes four read-only Boolean properties that represent these different trace levels. These properties are:

- *TraceSwitch.TraceError*
- *TraceSwitch.TraceWarning*
- *TraceSwitch.TraceInfo*
- *TraceSwitch.TraceVerbose*

-When a specified level is set, it and all lower levels return true.

5.2.8. Trace switches are used by Trace statements to test whether or not to write Trace output.

```
  // This example assumes the existence of a BooleanSwitch object named
  // myBooleanSwitch and a TraceSwitch object named myTraceSwitch
Trace.WriteIf(myBooleanSwitch.Enabled == true, "Error");
Trace.WriteLineIf(myTraceSwitch.TraceInfo == true, "Type Mismatch Error");
```

5.2.9. Trace switches can be configured after the application has been compiled by setting appropriate values in the `application .config` file. The switch `displayName` is utilized to identify the switch, for `BooleanSwitch` 0 = false, other = true. *TraceSwitch* receives the integers equivalent to the *TraceLevel* enum.

```
ApplicationName.exe.config
<system.diagnostics>
   <switches>
      <add name="Switch1" value="0" />
      <add name="Switch2" value="3" />
   </switches>
</system.diagnostics>
```

5.3. Creating a Unit Test Plan

5.3.1. Unit testing is the process by which units of your application are tested to ensure that they exhibit appropriate behavior. You test your units by creating test cases. A test case is a set of run-time conditions that tests a given path and parameters of execution.

5.3.2. You should design test cases to test a variety of different conditions. At the bare minimum, you should create test cases that test every possible data path through your unit. Additionally, you should test a variety of different data conditions including

- Normal data
- Boundary conditions
- Bad data
- Combinations of the preceding data conditions

5.4. Handling and Throwing Exceptions

5.4.1. Exceptions are how the common language run time reports run-time errors. When an exception is thrown, it is passed up the call stack until it finds an exception handling structure. If no such structure is found, an unhandled exception is handled by the run time's default exception handler, which leads to program termination.

5.4.2. You can design routines to handle exceptions that can allow an application to recover from an unexpected error, or at least terminate gracefully. An exception handling structure can consist of up to three parts:

try block
catch block
finally block

5.4.3. After an exception is handled, control is returned to the method that the exception handling routine resides in.

```
public void Parse(string aString)
{
   try
   {
      double aDouble;
      aDouble = Double.Parse(aString);
   }
   catch (System.ArgumentNullException e)
   {
      // In this block, you would place any code that should be
      // executed if a System.ArgumentNullException is thrown.
   }
```

```
   catch (System.Exception e)
   {
      // This block will catch any other exceptions that might be
      // thrown.
   }
   finally
   {
      // Any code that must be executed goes here. This code will be
      // executed whether an exception is thrown or not.
   }
}
```

5.4.4. Here you run a piece of code even in the event of an exception, which it is bubbled up.

```
try
{
   // Method code goes here
}
finally
{
   // Cleanup code goes here
}
```

5.4.5. You can use the *throw* keyword inside a *catch* block to rethrow the exception that is currently being handled. This passes the exception up the call stack. You can also wrap the current exception in a new exception by setting the new exception's *InnerException* property.

```
//Throwing exceptions
catch (System.NullReferenceException e)
{ throw e;}
//Or
catch
{ throw;}
//Wrapping and throwing the exception with additional information
catch (NullReferenceException e)
{
   throw new NullReferenceException("Widget A is not set", e);
}
```

5.4.6. You can create and throw custom exceptions by inheriting from the *System.ApplicationException* class. Exceptions should not be used for communication between components and clients, but should be reserved only for exceptional circumstances. If a condition occurs in your component that cannot be resolved, it is proper to throw an exception to the client application.

```
public class WidgetException:System.ApplicationException
{
   // This variable holds the Widget
   Widget mWidget;
   public Widget ErrorWidget
   {
      get
      {
         return mWidget;
      }
   }
   // The constructor takes a Widget, and a string that can be used
   // to describe program conditions at the time of the error. You can
   // provide different overloads of the constructor to take different
   // sets of parameters
   //
   // Declares the constructor for the class and calls the base class
```

```
    // constructor that sets the Message property inherited from
    // ApplicationException
    public WidgetException(Widget W, string S) : base(S)
    {
        // Sets the Widget property
        mWidget = W;
    }
}

//To throw this exception do the following
Widget Alpha;
// Code that corrupts Widget Alpha omitted
throw new WidgetException(Alpha, "Widget Alpha is corrupt!");
```

6. Data access using ADO.NET

6.1. Overview of ADO.NET

6.1.1. ADO.NET is a data access technology that is primarily disconnected and designed to provide efficient, scalable data access.

6.1.2. The Dataset is a disconnected, in-memory copy of part or all of a database. It contains

6.1.2.1. One or more DataTable objects, which contain

6.1.2.1.1. A DataColumn collectiom

6.1.2.1.2. A Constraint collection

6.1.2.1.3. A DataRows collection

6.1.2.2. A DataRelations collection wich enumerates a set of *DataRelation* objects that define the relations between tables in the DataSet

6.1.2.3. An ExtendedProperties collection that stores custom information about the DataSet.

6.1.3. The Data Provider is a set of classes that provide access to databases. The main components of the Data Providers are

- Connection object – provides connection to the database. (uses the ConnectionString property)
- Command object – Execute commands against a datasource (SQL) using the following methods:
  ExecuteNonQuery (for Insert/Delete/Update/Create/Alter)
  ExecuteScalar (Returns a single value)
  ExecuteReader (Returns a result set by a way of a DataReader object)
  ExecuteXmlReader (exclusive for sqlCommand)
- DataReader object – Provides a forward-only, read-only, connected recordset. Created only by a *ExecuteReader* method of the *Command* object
- DataAdapter object – Populates a disconnected *DataSet* or *DataTable* and performs updates. It provides 4 properties that represent database commands: *SelectCommand, InsertCommand, DeleteCommand, and UpdateCommand*.

6.1.4. Data Access in ADO.NET works as follows;

- The *Command* Object and the *DataAdapter* object use the *Connection* object to access the database.
- When using the *Command* Object to query a database, if there are results they will be sent to the *DataReader* object, which can be accessed by the logic of the program.
- The *DataAdapter* is used to populate a *Dataset* and to update the database when changing data in the *DataSet*.
- Updates to the database can be made through the *Command* object (SQL) or the *DataAdapter*.

6.1.5. Visual Studio .NET includes two Data Providers:

6.1.5.1. The *SQLDataProvider*, which contains classes optimized for accessing Microsoft SQL Server 7.0 or later.

6.1.5.2. The *OleDbDataProvider*, which contains classes that provide access to a broad range of database formats.

6.2. Accessing Data.

6.2.1. Connection Object

The *Connection* object connects to a database. You can create a *Connection* object by dragging a connection from the Server Explorer to the designer or by creating a new *Connection* object.

```
  // Declares and instantiates a new OleDbConnection object
OleDbConnection myConnection = new OleDbConnection();
```

```
  // Sets the connection string to indicate a Microsoft Access database at the specified
path
myConnection.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;DataSource=" +
                                                        "C:\\Northwind\\Northwind.mdb";
```

6.2.2. Command Object

The *Command* object represents a SQL command or a reference to a stored procedure in the database.

6.2.2.1. Methods.

Three methods for executing database commands are shared by OleDbCommand and SqlCommand objects:

- ExecuteNonQuery
- ExecuteScalar
- ExecuteReader

6.2.2.2. Properties. The command object is configured by setting these properties:

- *Connection*. An active C*onnection* object of the appropriate type.
- *CommandType*. Enumerator that can be:
  Text <default> (the CommandText has one or more SQL statement separated by a ;)
  StoredProcedure (The CommandText has the Stored Procedure name)
  TableDirect (The CommandText has the name of table or tables)
- *CommandText*

6.2.2.3. Parameters

Parameters represent values required for the execution of commands represented by *Command* objects. The *OleDbCommand* object uses a question mark (?) as a placeholder for parameters in SQL statements, whereas the *SqlCommand* object uses named parameters.

```
SELECT EmpId, Title, FirstName, LastName FROM Employees
WHERE (FirstName = ?) AND (LastName = ?)
SELECT EmpId, Title, FirstName, LastName FROM Employees
WHERE (Title = @Title)
```

6.2.2.3.1. Parameters are stored in the Command's object *Parameters* property, and they are instances of the *OleDbParameter* or *SqlParamete* class. The parameter object performs datatype conversion between the Commont Type System (CTS) and the database datetypes. Some of its properties are:

- DbType (This property is not visible in the designer.)
- Direction (Input, output or return value of a stored procedure)
- OleDbType (OleDbParameters only)
- ParameterName (name to use to access the parameter)
- Precision (maximum number of digits)
- Scale (maximum number of decimal places)
- Size (Maximum size for binary and string parameters)
- SourceColumn (Column used to look up or map values)
- SourceVersion (version that is being edited)
- SQLType (SQLParameters only)
- Value (value represented by the parameter)

```
  // This line sets the value by referring to the index of the parameter
OleDbCommand1.Parameters[0].Value = "Hello World";
  // This line sets the value by referring to the name of the parameter
OleDbCommand1.Parameters["myParameter"].Value = "Goodbye for now";
  // This command is identical whether you are using the OleDbCommand/SqlCommand class
myCommand.ExecuteNonQuery();
  // This command is identical whether you are using the OleDbCommand/SqlCommand class
Object O;
O = myCommand.ExecuteScalar();
```

6.2.2.3.2. Example of executing Ad Hoc queries

```
// This example assumes a connection named myConnection. It also
// assumes using System.Data.OleDb
public void DeleteRecord(string aString)
{
   string Cmd;
   Cmd = "DELETE * FROM Employees WHERE Name = '" + aString + "'";
     // Specifies Cmd as the command string and myConnection as the connection
   OleDbCommand myCommand = new OleDbCommand(Cmd, myConnection);
     // Opens the Connection and executes the command
   myConnection.Open();
   myCommand.ExecuteNonQuery();
```

```
     // Always close the connection
   myConnection.Close();
}
```

6.2.3. DataReader

6.2.3.1. *DataReader* objects provide forward-only, read-only, connected data access and require the exclusive use of a data connection.

```
  // This example assumes the existence of an OleDbCommand and a SqlCommand objects
  // named myOleDbCommand and mySqlCommand respectively
System.Data.OleDb.OleDbDataReader myOleDbReader;
System.Data.SqlClient.SqlDataReader mySqlReader;
  // This call creates a new OleDbReader and assigns it to the variable
myOleDbReader = myOleDbCommand.ExecuteReader();
  // This call creates a new SqlReader and assigns it to the variable
mySqlReader = mySqlCommand.ExecuteReader();
```

6.2.3.2. *DataReader* objects expose methods that allow retrieval of strongly typed data.

```
  // Opens the active connection
myConnection.Open();
  // Creates a DataReader and assigns it to myReader
System.Data.OleDb.OleDbDataReader myReader = myOleDbCommand.ExecuteReader();
  // Calls Read before attempting to read data
while (myReader.Read())
{
     // You can access the columns either by column name or by ordinal number
   Console.WriteLine(myReader["Customers"].ToString());
   object myObject = myReader[3];
    // Use a "Get" method to retrieve string type data
   string myString = myReader.GetString(3);
    //Get the ordinal for a column to use a "Get" method
  int IndexValue = myReader.GetOrdinal("CustomerID");
  string Customer = myReader.GetString(IndexValue);
}
  // Always close the DataReader when you are done with it
myReader.Close();
  // And close the connection if not being used further
myConnection.Close;
```

6.2.3.3. To read from a *DataReader* with multiple result sets (due to several sql statements separated by ;), use the *NextResult* method. However this method comes postionated at the first result set (no like the read method that comes positionated before the first data). Do not call *NextResult* at the beginning or you will lose the first result set:

```
Do
{
   while (myReader.Read())
   {
      // Add code here to loop through the records of the current result set
   }
} while (myReader.NextResult());
```

6.3. Using DataSet Objects and Updating Data

6.3.1. *DataAdapter* objects facilitate interaction between a database and a *DataSet* by managing the commands required to fill the *DataSet* from the database and update the database from the *DataSet*.

6.3.2. Typed DataSet objects are instances of classes derived from the *DataSet* class that are based on an XML schema and expose strongly typed data and member tables, and columns with friendly names. You can only create a typed DataSet when you know the structure of the data you will be working with in advance.

```
  //Each example returns the OrderID Column from the first record of the orders table
  //of the dsOrders recordSet untyped data set
string myOrder;
myOrder = (string)dsOrders.Tables["Orders"].Rows[0]["OrderID"];
  //Strongly typed data set
myOrder = dsOrders.Orders[0].OrderID;
```

6.3.3. You can create *DataSet* objects and *DataTable* objects independent of *DataAdapter* objects and fill them programmatically.

```
  //Create the dataset object
DataSet myDataSet = new DataSet();
  //Create a new DataTable and add it to the Tables collection
DataTable myTable = new DataTable();
myDataSet.Tables.Add(myTable);
  //Create new columns and add them to the Columns collection of the DataTable
```

```
DataColumn AccountsColumn = new DataColumn("Accounts");
myDataSet.Tables[0].Columns.Add(AccountsColumn);
  //To add a row, first call the NewRow method of the DataTable you are adding the DataRow
DataRow myRow;
myRow = myDataSet.Tables[0].NewRow();
  //Populate the datarow, in this case will be with the members of a collection of strings
  // This example assumes the existence of an ArrayList object named StringCollection
for (int i = 0; i < StringCollection.Count; i ++)
              myRow.Item[Counter] = StringCollection[i];
  //After the DataRow has been populated, you can add it to the
  //Rows collection of the DataTable
myDataSet.Tables[0].Rows.Add(myRow);
```

6.3.4. You can read and access text files using the *System.IO.StreamReader* class and parse them with methods of the *String* class.

```
// This example assumes the existence of a text file named myFile.txt
// that contains an undetermined number of rows with seven entries
// in each row delimited by commas.
//Creates a new DataSet
DataSet myDataSet = new DataSet();
  // Creates a new DataTable and adds it to the Tables collection
DataTable aTable = new DataTable("Table 1");
myDataSet.Tables.Add("Table 1");
  // Creates and names seven columns and adds them to Table 1
DataColumn aColumn;
for (int counter = 0; counter < 7; counter ++)
{
   aColumn = new DataColumn("Column " + counter.ToString());
   myDataSet.Tables["Table 1"].Columns.Add(aColumn);
}
  // Creates the StreamReader to read the file and a string variable to
  // hold the output of the StreamReader
System.IO.StreamReader myReader = new
   System.IO.StreamReader("C:\\myFile.txt");
string myString;
  // Checks to see if the Reader has reached the end of the stream
while (myReader.Peek() != -1)
{
     // Reads a line of data from the text file
   myString = myReader.ReadLine();
     // Uses the String.Split method to create an array of strings that
     // represents each entry in the line. That array is then added as
     // a new DataRow to Table 1
   myDataSet.Tables["Table 1"].
      Rows.Add(myString.Split(char.Parse(",")));
}
```

6.3.5. *DataRelation* objects represent parent-child relationships between columns of different tables. You can use *DataRelation* objects to enforce constraints and retrieve related rows of data.

```
  //Create a new datarelation object
DataRelation CustomersOrders = new DataRelation("Data Relation 1", column1, column2);
  //Add the object to the Relations collection of a dataset before it becomes active
myDataSet.Relations.Add(CustomersOrders);
  //Retrieving using the datarelation CustomerOrders in the datset myDataset
DataRow[] ChildRows; //Array of rows
DataRow ParentRow;
  // This returns all rows that have a child relationship to Row 1 of
  // the Customers table as defined by the CustomersOrders DataRelation
ChildRows = myDataSet.Tables["Customers"].Rows[1].GetChildRows(CustomersOrders);
  // This returns the row that has a parent relationship to row 5 of
  // the Orders table as defined by the CustomersOrders DataRelation
ParentRow = myDataSet.Tables["Orders"].Rows[5].GetParentRow(CustomersOrders);
```

6.3.6. *Constraints* are rules that represent how data can be added to particular columns. There are two kinds of constraints: the *UniqueConstraint* (to define the primary key), and the *ForeignKeyConstraint*. They are enforced only if the *EnforceConstrains* property of the Dataset is set to true.

6.3.6.1. A *ForeignKeyConstrain* has 3 properties that define rules to be enforced in the parent-child relationship

- UpdateRule. Enforced when parent row is updated
- DeleteRule. When parent row is deleted
- AcceptRejectRule. When the AcceptChanges method from the table is called.

(can be set to cascade or none)

6.3.6.2. Those properties can be set to the following

- Cascade. Changes in the parent are cascaded to the children rows.
- None. No changes are cascaded. It can result in invalid references in children rows.
- SetDefault. The foreing key in the child row is set to its default value (from the default property)
- SetNull. The foreing key in the child row is set to null. It can result in invalid data

6.3.6.3. Ways of creating Unique constrains for a table.

```
  //Setting the Unique property of a DataColumn to "true",creates a
  //UniqueConstraint behind the scenes
myDataColumn.Unique = true;
  //creating a new UniqueConstraint that specifies a column to be unique
  //and add it to the Constraints collection manually
UniqueConstraint myConstraint = new UniqueConstraint(myDataColumn);
myDataTable.Constraints.Add(myConstraint);
  //can specify an array of columns that must contain a unique
  //combined value with a UniqueConstraint
DataColumn[] myColumns = new DataColumn[2];
myColumns[0] = EmployeesTable.Columns["FirstName"];
myColumns[1] = EmployeesTable.Columns["LastName"];
UniqueConstraint myConstraint = new UniqueConstraint(myColumns);
EmployeesTable.Constraints.Add(myConstraint);
```

6.3.6.4. Ways of creating a foreing constrain

```
  //Specify parent and child column
ForeignKeyConstraint myConstraint = new
           ForeignKeyConstraint(CustomersTbl.Columns["CustomerID"],
                                OrdersTbl.Columns["CustomerID"]);
  //They become active after being added to the constrains collection of the parent table
CustomersTbl.Constraints.Add(myConstraint);
```

6.3.7. A DataSet maintains two versions of data: the original version and a version that includes any modifications. When the *Update* method of the *DataAdapter* is called, the original version of the data is compared to the updated version and used to generate the commands needed for the update.

6.3.8. Each *Datarow* contains 2 versions: un original unedited version and another edited with changes. You can reject or commit those changes:

```
  //Modifying programmatically a column
myDataRow[2] = "Splunge";
myDataRow["Customers"] = "Winthrop";
  //Reverting a row to its original state
myDataRow.RejectChanges();
  //Commiting changes (Changes won't be reflected in the database)
myDataRow.AcceptChanges();
```

6.3.9. To determine the state of a *datarow* check the RowsState property. Its values are:

- Unchanged
- Modified
- Added (new rows added to the DataRowCollection)
- Deleted (Rows have been deleted with DataRow.Delete)
- Detached (The row has been created but is not part of any DataRowCollection)

6.3.10. The database can be updated through dataAdapters

```
  //Two dataAdapters update from the same dataset
myDataAdapter.Update();
myOtherDataAdapter.Update();
  //Specify the particular dataset, DataTable or DataRows to update
myDataAdapter.Update(myDataSet);
myDataAdapter.Update(myDataTable);
myDataAdapter.Update(myDataRows);
```

6.3.11. Transactions

6.3.11.1. You can execute multiple commands transactionally by obtaining a *Transaction* object from an open connection and assigning the *Transaction* property of the *Command* objects to that reference. After the commands have been executed, all commands included in the transaction can be committed or rolled back.

```
// Transactions should be enclosed in a Try..Catch..Finally block to
// catch any exceptions that might occur
System.Data.OleDb.OleDbTransaction myTransaction = null;
try
{
   myConnection.Open();
     // This creates a new transaction object and assigns it to myTransaction
   myTransaction = myConnection.BeginTransaction();
```

```
     // Adds Update1 and Update2 to the transaction.
   Update1.Transaction = myTransaction;
   Update2.Transaction = myTransaction;
     // Executes Update1 and Update2
   Update1.ExecuteNonQuery();
   Update2.ExecuteNonQuery();
     // If no exceptions occur, commits the transaction
   myTransaction.Commit();
}
catch (Exception ex)
{
     // The transaction is not executed if an exception occurs
   myTransaction.Rollback();
}
finally
{
     // Whether an exception occurs or not, the connection is then closed
   myConnection.Close();
}
```

6.3.11.2. To use transactions with *DataAdapter* objects first create a transaction with the *BeginTransaction* method of an open connection. You must then assign that transaction to the *Transaction* property of the *InsertCommand, UpdateCommand*, and *DeleteCommand* of each *DataAdapter* that will be involved in the database update. You can then call the *Update* method of each *DataAdapter* and call the *Commit* or *Rollback* method of the transaction as necessary.

6.3.12. Handling update errors

6.3.12.1. You can handle update errors in the *RowUpdated* event handler of the data adapter. This event fires after a row update has been attempend and before any exception. You can determine if an error has occurred by examining the event argument's *Status* property and can handle errors by setting the *Status* property to an appropriate value.

6.3.12.2. The *RowUpdated* event provides an instance of the *slqRowUpdateEventArgs* or *OleDbRowUpdateEventaArgs*. The argument has the following properties:

Command	Represents the command to execute when performing the update.
Errors	Returns any errors generated by the Data Provider when the command executes.
RecordsAffected	Number of records affected by the command represented by the Command property.
Row	Returns the row that was updated.
Status	Returns the UpdateStatus of the command. It can have the following values - *Continue*. The DataAdapter continues processing rows.(default when no errors) - *ErrorsOccurred*. Errors occurred while attempting to update this row - *SkipAllRemainingRows*. Skip current and remaining updates. - *SkipCurrentRow*. Skip update of current row, proceed with the rest of the updates.

6.3.12.3. Handling update errors in the *RowUpdate* event handler of a *SqlDataAdapter*

```
private void myDataAdapter_RowUpdated(object sender,
                                      System.Data.SqlClient.SqlRowUpdatedEventArgs e)
{
     // Checks the event arguments Status property to see if an error occurred
   if (e.Status == UpdateStatus.ErrorsOccurred)
   {
        // Informs the user that an error occurred and gives some information about it
      MessageBox.Show("An error of type " + e.Errors.ToString() +
                           "occurred. Here is some additional information: " +
                            e.Errors.Message);
        // Skips the update for this row but proceeds with the rest
      e.Status = UpdateStatus.SkipCurrentRow;
   }
}
```

6.4. Binding, Viewing, and Filtering Data

6.4.1. Binding

6.4.1.1. Data binding refers to the relationship between a data provider and a data consumer. Data providers make data available, and data consumers receive data and display or otherwise process it. Any object that implements the *Ilist* interface can be a data provider.

6.4.1.2. There are two kinds of data binding: simple binding and complex binding. A simple-bound control can only bind a single record at a time, whereas a complex-bound control binds all available records at once.

6.4.1.3. Data binding for controls is managed through the control's *DataBindings* property, which is an instance of a *ControlBindingsCollection* object. Any run-time available property of a control can be bound to a data source.

```
//Adding data bindings requieres the property name you want to bind (String);
//the data source you want to bind to (an object); and the data member of
//the data source you want to bind the property to.

  TextBox1.DataBindings.Add("Text", DataSet1.Customers, "CustomerID");

//Binding to an object that doesn't have multiple data members,
//such as a collection or array.

  String[] myStrings = new String[3];
  myStrings[0] = "A";
  myStrings[1] = "String";
  myStrings[2] = "Array";
  TextBox1.DataBindings.Add("Text", myStrings, "");

//This method requires a Binding object as a parameter, which you can
//also access through the DataBindings property.

  Label1.DataBindings.Remove(Label1.DataBindings["Text"]);

//You can remove all data bindings from a control by calling the
//DataBindings.Clear

  Label1.DataBindings.Clear();
```

6.4.1.4. Data currency is managed by *CurrencyManager* objects. Every data source has an associated *CurrencyManager* that keeps track of the current record. *CurrencyManager* objects are in turn managed by the *Form.BindingContext* property.

```
  //To access a data source's currency manager supply to the
  //BindingCOntext property the data source name
this.BindingContext[DataSet1.Customers]
  //Set the current record by setting the BindingContext's "Position" property
  // The following examples assume a Customers table of a DataSet named
  // DataSet1 that is resident on the current Windows form
  // Sets the current record to the first record of the data source
this.BindingContext[DataSet1.Customers].Position = 0;
  // Advances the current record by one
this.BindingContext[DataSet1.Customers].Position ++;
  // Moves the current record back one
this.BindingContext[DataSet1.Customers].Position --;
  // Sets the current record to the fifth record in the data source
this.BindingContext[DataSet1.Customers].Position = 4;
  // Advances to the last record
this.BindingContext[DataSet1.Customers].Position =
                    DataSet1.Tables["Customers"].Rows.Count - 1;
```

6.4.1.5. Example of disabing navigation buttons utilizing the BindingContext's *position* property

```
  // This adds a method to handle the PositionChanged event
public void OnPositionChanged(object sender, System.EventArgs e)
{
     // Checks to see if the CurrencyManager is at the start of the records
   if (this.BindingContext[DataSet1.Customers].Position == 0)
        // Disables the back button
      BackButton.Enabled = false;
   else
        // Enables the back button
      BackButton.Enabled = true;
     // Checks to see if the CurrencyManager is at the end of the records
   if (this.BindingContext[DataSet1.Customers].Position ==
                                        DataSet1.Tables["Customers"].Rows.Count -1)
        // Disables the forward button
      ForwardButton.Enabled = false;
   else
        // Enables the forward button
      ForwardButton.Enabled = true;
}

// You must also hook up the event to the method that is to handle
// it by adding the following line to the Form's constructor

this.BindingContext[DataSet1.Customers].PositionChanged += new
   EventHandler(this.OnPositionChanged);
```

6.4.1.6. In complex binding more than a record can be bound at a time.

```
  //In a datagrid
```

```
DataGrid1.DataSource = DataSet1.Customers;

  //ListBox, CheckedListBox, and ComboBox can display just a column.
ComboBox1.DataSource = DataSet1.Customers;
ComboBox1.DisplayMember = "CustomerID";
```

6.4.2. DataView

6.4.2.1. A *DataView* is an object that is associated with a *DataTable* and provides a filterable, sortable subset of the data contained by the underlying table. You can create a *DataView* in code or by dragging a *DataView* object and setting its *table* property.

```
   //Creating giving the table parameter
DataView myDataView = new DataView(myDataTable);
  //Creating and later setting the Table property
DataView myDataView = new DataView();
myDataView.Table = myDataTable;
```

6.4.2.2. The *RowFilter* and *Sort* properties of *DataView* objects are used to specify sorting and filtering conditions for a particular *DataView* object.

```
  //Provide the column(s) to be used in sorting expression
myDataView.Sort = "CustomerID";
myDataView.Sort = "State, City";
  // In this example the rows will be sorted by descending state, but
  // ascending city, as sorting is ascending unless otherwise marked
  myDataView.Sort = "State DESC, City";

  //To Filter provide an expression with SQL syntax.
  //String must be enclosed by quotes (') and dates by #.
myDataView.RowFilter = "City = 'Seattle'";
myDataView.RowFilter = "City = 'Seattle' AND State = 'WA'";
myDataView.RowFilter = "City = 'Seattle' OR State = 'WA'";
myDataView.RowFilter = "City = 'Des Moines' AND (NOT State = 'IA')";
myDataView.RowFilter = "Length >= 10 AND Height < 4";
myDataView.RowFilter = "CityState = City + 'WA'";
myDataView.RowFilter = "Price * 1.086 <= 500";
myDataView.RowFilter = "City IN('Seattle', 'Tacoma', Blaine')";
  // For string comparisons, * is a wildcard that stands for any single
  // character, % stands for any number of any characters.
myDataView.RowFilter = "City LIKE 'Se*t%e'";
```

6.4.2.3. Use the *DataView.RowState* property to filter *DataRow* objects based in their state. You can set this property to more than one value. Posible values are:

Setting	Description
Unchanged	Displays rows that have not been changed.
Added	Displays rows that have been added since the last DataSet update.
Deleted	Displays rows that have been deleted since the last DataSet update.
OriginalRows	Original rows including unchanged and deleted rows.
CurrentRows	Current rows including added, modified, and unchanged rows.
ModifiedCurrent	A current version, which is a modified version of original data.
ModifiedOriginal	The original version (although it has since been modified and is available as ModifiedCurrent).

6.4.2.4. Dataview are fully editable. Change any of the following *DataView* properties to restrict edition:

- AllowDelete.
- AllowEdit.
- AllowNew

6.4.2.5. A *DataViewManager* manages *DataView* objects for an entire *DataSet*. Individual *DataView* object properties can be set through the *DataViewSettings* property, and individual *DataView* objects can be retrieved with the *CreateDataView* method.

```
  //Creation with the dataset as a parameter
DataViewManager myDataViewManager = new DataViewManager(myDataSet);
  //Creation and assigning the dataset property
DataViewManager myOtherDataViewManager = new DataViewManager();
myOtherDataViewManager.DataSet = myOtherDataSet;
```

6.4.2.6. The *DataViewSettings* property exposes a collection of *DataView* property values, one for each table in the dataset and it is utilized to manage *RowFilter*, *Sort* and other properties

```
  //Set the RowFilter Property of the DataView Associated with the Customers table
myDataViewManager.DataViewSettings["Customers"].RowFilter = "State = 'WA'";
  //To retrieve a DataView from the DataViewManager use:
DataView myDataView;
myDataView = myDataViewManager.CreateDataView(DataSet1.Tables[0]);
```

6.5. Using XML in ADO.NET

6.5.1. XML is the behind-the-scenes foundation of ADO.NET. Data can be represented in memory or in a file as XML.

6.5.2. An instance of an *XmlReader* can be obtained by executing a *SqlCommand* that contains a SELECT command that includes a valid FOR XML clause (Only SQL server 2000). It is created by using a *sqlCommand.ExecuteXMLReader*. The *XmlReader* provides connected, read-only, forward-only access to a database in XML format. Its behavior is similar to a *DataReader*

```
  // This example assumes the existence of a valid SqlConnection named SqlConnection1.
System.Xml.XmlReader myReader;
SqlCommand mySQLCommand = new SqlCommand(
                     "SELECT * FROM Customers FOR XML AUTO, XMLDATA", SqlConnection1);
SqlConnection1.Open();
myReader = mySQLCommand.ExecuteXmlReader();
while (myReader.Read())
{
     // Writes the content, including markup, of this node and any child nodes to the Console
   Console.WriteLine(myReader.ReadOuterXml());
}
myReader.Close();
SqlConnection1.Close();
```

6.5.3. The *DataSet* object can read and write data formatted as XML. The *ReadXml* and *WriteXml* methods are used to load XML into a *DataSet* and write data in XML format, respectively. DataSets can also read and write XML schemas with the *ReadSchem*a and *WriteSchema* methods.

```
  //Read XML data from a file (also can be a XmlReader or TextReader object)
DataSet myDataSet = new DataSet();
myDataSet.ReadXml("C:\\myData.XML");
  //Read the structure of the data into the DataSet, not the data itself.
myDataSet.ReadXmlSchema("C:\\mySchema.XML");
  //Write the dataset contents to a file (also it can be a stream, XmlWriter or TextWriter)
myDataSet.WriteXml ("C:\\myData.XML");
  //Write the structure of the data as squema
myDataSet.WriteXmlSchema("C:\\mySchema.XML");
```

6.5.4. The *XmlDataDocument* is an in-memory representation of an XML document that is synchronized with a DataSet. Any changes made to the DataSet are directly transmitted to the *XmlDataDocument* and vice versa.

```
  //Creating an XmlDataDocument with its associated DataSet.
  //The data and squema of the Dataset are loaded into the XmlDataDocument
XmlDataDocument myDocument = new XmlDataDocument(myDataSet);
  //When no associated dataSet is specified, and empty dataSet is created
XmlDataDocument myDocument = new XmlDataDocument();
  //In that case the empty dataset has to read a squema from a XML file,
  //XML stream, XmlReader, or a TextReader
myDocument.DataSet.ReadXmlSchema("C:\\myXml.xml");
  //Calling the Load method to load the XML data.
myDocument.Load("C:\\myXml.xml");
```

6.5.5. XSLT transforms are used to transform XML from one format to another. Transforms can be executed on an *XmlDataDocument* and require an XSLT style sheet that contains the definition for the transformation.

```
  //Create a XslTransform object with a file containing a style sheet
Xml.Xsl.XslTransform myTransform = new Xml.Xsl.XslTransform();
myTransform.Load("C:\myStyle.xsl");

  //Transformation parameters;
  // Object to transform (an XmlDataDocument or an object implementing
  // IXPathNavigable)
```

```
  // An instance of the System.Xml.Xsl.XsltArgumentList with parameters
  // required by the style sheet or null
  //The ouput object (a Stream, a TextWriter, or an XmlWriter)


  // The StreamWriter will receive the output from the transform and
  // write it to a text file
System.IO.StreamWriter
myWriter = new System.IO.StreamWriter("myTextFile.txt");

myTransform.Transform(myDocument, null, myWriter);
```

7. Creating Controls Using the .NET Framework

7.1. Using GDI+

7.1.1. The Graphics object is the principal object used in rendering graphics. It represents a drawing surface and provides methods for rendering to such surface.

7.1.2. GDI+ (Graphical Device Interface) is the name given to the .NET framework used to display graphical information on the computer screen and it is wrapped into 6 different namespaces:

- System.Drawing. The most utilized for graphic programminging.
- System.Drawing.Design. Classes that are designed to extend the design time user interface.
- System.Drawing.2D. Classes that are designed to render advanced visual effects.
- System.Drawing.Imaging Classes that allow advanced manipulation of image files.
- System.Drawing.Printing Classes that facilitate printing content.
- System.Drawing.Text Classes that facilitate advanced manipulation of fonts.

7.1.3. All the visual elements are rendered using a *Graphics* object. You can get this object from a visual element. For classes inheriting from *Control* use the *CreateGraphics* method.

```
System.Drawing.Graphics myGraphics;
  //Here the graphics object can be used to render graphics in the form
myGraphics = myForm.CreateGraphics();
  //Here you create a bitmap object out of a file and get its graphics object
Bitmap myImage = new Bitmap("C:\\myImage.bmp");
myGraphics = Graphics.FromImage(myImage);
```

7.1.4. Position in the screen starts from the upper left corner. The *System.Drawing* namespace has the following structures to describe locations

Point. A single Integer x,y point
PointF. A single float x,y point
Size. Rectangular height and Width size using integer
SizeF. Rectangular height and Width size using floats
Rectangle. A rectangle with dimension and position on integers
RectangleF. A rectangle with dimension and position on floats

```
  //Defining point structures
Point myPoint;
PointF myPointF = new PointF(13.5F,33.21F);
myPoint = new  Point((int)myPointF.X, (int)myPointF.Y);


  //Defining a Rectangle
Point myOrigin = new Point(10, 10);
Size mySize = new Size(20, 20);
  // Creates a 20 by 20  rectangle with point(10,10) as the upper
  // left corner
Rectangle myRectangle = new  Rectangle(myOrigin, mySize);
```

7.1.5. Pens, Brushes, and Colors are objects that are used to control how a graphic is rendered to a drawing surface. Pens draw lines, Brushes fill shapes, and Colors structures represent colors. You can use *SystemPens*, *SystemBrushes*, and S*ystemColors* to maintain a coherent appearance with the rest of the application. The *Graphics* object has a wide variety of methods to render shapes to the screen.

- "Draw" are used to draw lines and structures. They use a *Pen* object.
- "Fill" are used to render solid shapes. They use a *Brush* object.
  (*SolidBrush*, *TextureBrush*, *HatchBrush*, *LinearGradientBrush*, and *PathGradientBrush)*.

```
  //Set Alpha(tranparency), red, green, blue
Color myColor;
```

```
myColor = Color.FromArgb(128, 255, 12, 43);
  //Or you can omit Alpha
myColor = Color.FromArgb(255, 12, 43);
  //or you can set a predefined color
myColor = Color.Tomato;
  //Get a system color
Color myColor = SystemColors.HighlightText;

  //Creating a Brush
SolidBrush myBrush = new SolidBrush(Color.PapayaWhip);
  //create a pen with a default width of one
Pen myPen = new Pen(Color.BlanchedAlmond);
  //Now specifying the widht
Pen myPen = new Pen(Color.Lime, 4);
  //create a pen from a preexisting brush
Pen myPen = new Pen(myBrush);
  //You can also use SystemPens and SystemBrushes.
```

7.1.6. Simple shapes and text can be rendered using the methods provided by the *Graphics* object.

```
  // Creates the Rectangle object
Rectangle myRectangle = new Rectangle(0, 0, 30, 20);
  // Creates the Graphics object that corresponds to the form
Graphics g = this.CreateGraphics();
  // Uses a system pen to draw the rectangle)
g.DrawRectangle(SystemPens.ControlDark, myRectangle);
  // Always disposes the Graphics object
g.Dispose();

//Code inside of a form
SolidBrush myBrush = new SolidBrush(Color.MintCream);
Graphics g = this.CreateGraphics();
  // The ellipse will be inscribed within the rectangle
Rectangle myRectangle = new Rectangle(0, 0, 30, 20);
g.FillEllipse(myBrush, myRectangle);
  // Dispose the Graphics object and the Brush
g.Dispose();
myBrush.Dispose();

  //To render text use the Graphics.DrawString menthod
  // This example uses the SystemBrush class to supply one of the
  // system brushes
Graphics g = this.CreateGraphics();
String myString = "Hello World";
Font myFont = new Font("Times New Roman", 36, FontStyle.Regular);
  // The final two parameters are X and Y coordinates
g.DrawString(myString, myFont, SystemBrushes.Highlight, 0, 0);
  // Always dispose your Graphics object
g.Dispose();
```

7.1.7. Complex shapes should be defined in terms of a *GraphicsPath* object, (System.Drawing.Drawing2D) which exposes a variety of methods to facilitate defining complex shapes. When complete, a GraphicsPath object can be rendered by the *Graphics* object.

7.1.7.1. Adding figures

The array of points provides coordinates to map the path to, and the array of bytes describes what kind of line passes through the points.

```
// This example assumes using System.Drawing.Drawing2D
GraphicsPath myPath = new GraphicsPath(new Point[] {new Point(1, 1),
   new Point(32, 54), new Point(33, 5)}, new byte[] {
   (byte)PathPointType.Start, (byte)PathPointType.Line,
   (byte)PathPointType.Bezier});
```

An then you can add figures using:

Method	Description
AddClosedCurve	Adds a closed curve described by an array of points to the *GraphicsPath*.
AddEllipse	Adds an ellipse to the *GraphicsPath*.
AddPath	Adds a specified instance of *GraphicsPath* to the current path.

Method	Description
AddPie	Adds a pie shape to the *GraphicsPath*.
AddPolygon	Adds a polygon described by an array of points to the *GraphicsPath*.
AddRectangle	Adds a rectangle to the *GraphicsPath*.
AddRectangles	Adds an array of rectangles to the *GraphicsPath* .
AddString	Adds a graphical representation of a string to the *GraphicsPath*, in the specified font

7.1.8. Adding lines elements

```
GraphicsPath myPath = new Drawing2D.GraphicsPath();
myPath.StartFigure();
// Insert code to add line elements to the figure here
myPath.CloseFigure();
```

To add line elements use

Method	Description
AddArc	Adds an arc to the current figure.
AddBezier	Adds a Bezier curve to the current figure.
AddBeziers	Adds a series of connected Bezier curves to the current figure.
AddCurve	Adds a curve described by an array of points to the current figure.
AddLine	Adds a line to the current figure.
AddLines	Adds a series of connected lines to the current figure.

7.1.9. In any case, finally Call *Graphics.DrawPath* to draw the outline of the path or *Graphics.FillPath* to draw a filled *GraphicsPath*

7.2. Authoring Controls

7.2.1. All controls inherit either directly or indirectly from the base class *Control*. The *Control* class provides keyboard and pointer functionality as well as a set of common control properties.

7.2.2. There are three primary approaches to control authoring

- Inheriting from an existing control.
- Creating a user control,
- Creating a custom control.

7.2.3. Public properties that are added to controls are automatically displayed in the Properties window at design time. To avoid that, mark the property with the *browsable* attribute and specify false.

```
[System.ComponentModel.Browsable(false)]
public int StockNumber
{
   // Property code omitted
}
```

7.2.4. Inherited control

You can create an inherited control by inheriting from an existing Windows Forms control. The new control has the same visual representation and functionality as the base control. There are 2 reasons to do that.

7.2.4.1. To add new functionality to an inherited control.

```
//Control that only takes number by overriding the OnKeyPress method
public class NumberBox : System.Windows.Forms.TextBox
{
   protected override void OnKeyPress(KeyPressEventArgs e)
```

```
    {
       if (char.IsNumber(e.KeyChar) == false)
          e.Handled = true;
    }
}
```

7.2.4.2. To create a new appearance for an existing control.

Override the *OnPaint* method and set the *Region* property. A *Region* is a class that describes a regular or irregular region of the screen and can be created from a *GrahicsPath* object.

```
//This button is rendered as a string of text
public class WowButton : System.Windows.Forms.Button
{
   protected override void OnPaint(PaintEventArgs pe)
   {
      System.Drawing.Drawing2D.GraphicsPath myPath = new
                                                  System.Drawing.Drawing2D.GraphicsPath();
        // This line sets the GraphicsPath to render a 72 point string
        // in the FontFamily and the FontStyle of the control.
        // you must cast Font.Style to an int.
      myPath.AddString("Wow!", Font.FontFamily, (int)Font.Style, 72,
                                          new PointF(0, 0), StringFormat.GenericDefault);
        // Creates a new Region from the GraphicsPath
      Region myRegion = new Region(myPath);
        // Assigns the new Region to the Region property of the control
      this.Region = myRegion;
   }
}
```

7.2.5. User Controls

7.2.5.1. A user control encapsulates one or more Windows Forms controls and inherits from the *UserControl* class. Constituent controls are private by default and to be access change the Modifiers property at design time or wrapi the controls properties inside of other Public properties. This is an example of custom code included inside the user control:

```
 //In this example each textBox control has its OnKeyPress event handler set
 //to add their numeric contents and be displayed in a Label control
protected override void OnKeyPress(object sender, KeyPressEventArgs e)
   {
     //  Verifies that a number was pressed
   if (char.IsNumber(e.KeyChar) == false)
      e.Handled = true;
   Label1.Text = (int.Parse(TextBox1.Text) +
      int.Parse(TextBox2.Text)).ToString();
}

  //Here the button's backcolor property is wrapped inside a Public property
  //to make it accessible to users of the control
public color ButtonColor
{
   get
   {
      return Button1.BackColor;
   }
   set
   {
      Button1.BackColor = value;
   }
}
```

7.2.6. Custom Controls

A custom control is the most time-consuming approach to control development. They inherit from the *Control* class and include no graphic representation.

To display the control implement the *Paint* method, which has a *PaintEventArgs* class argument. This class has the *Graphics* property (the drawing surface of the control) and the *ClipRectangle* property (the bounds of the control).

This example shows an *OnPaint* method that renders the custom control as a single red ellipse.

```
  // This example assumes using System.Drawing
protected override void OnPaint(PaintEventArgs e)
{
   Brush aBrush = new SolidBrush(Color.Red);
   Rectangle clientRectangle = new Rectangle(new Point(0,0), this.Size);
   e.Graphics.FillEllipse(aBrush, clientRectangle);
}
```

```
  //To allow resizing of the control put in the constructor
SetStyle(ControlStyles.ResizeRedraw, true);

  //Or call the Refresh method
Refresh();
```

7.3. Common Tasks Using Controls

7.3.1. You can add your controls to the Toolbox by right-clicking the Toolbox and choosing Customize Toolbox. Your controls will appear in the Toolbox with a default icon. If you want to specify a Toolbox bitmap for your control, use the *ToolboxBitmap* Attribute.

```
  //Specifying a file as the bitmap of the control
[ToolboxBitmap(@"C:\Pasta.bmp")]
public class PastaMaker : Control
{
   // Implementation omitted
}

  //Using a Type to assign the bitmap to the control
[ToolboxBitmap(typeof(Button))]
public class myButton : Button
{
   // Implementation omitted
}
```

7.3.2. To debug controls, they must first be built and hosted in a form. If your control is in an executable project, you can add a form to the project to serve as a test form. If your control is in a nonexecutable project, you must add a test project to host the form. Once hosted, you can use the regular Visual Studio debugging tools to debug your control. You must rebuild the control before changes will be incorporated.

7.3.3. The .NET Framework provides a licensing model for licensing your controls. Specify a *LicenseProvider* by applying the *LicFileLicenseProvider* attribute to the control. Call *LicenseManager.Validate* to retrieve a reference to the license. When finished, dispose the license. The validation is performed by examining a file named FullName.LIC where FullName is the fully qualified name of the control and searching for the text "myClassName is a licensed component." where myClassName is again the fully qualified name of the component.

```
  // This example assumes using System.ComponentModel
  // The LicenseProvider attribute specifies the kind of LicenseProvider to use.
[LicenseProvider(typeof(LicFileLicenseProvider))]
public class Widget : System.Windows.Forms.Control
{
   private License myLicense;
   public Widget()
   {
        // Validates and retrieves a reference to the license
      myLicense = LicenseManager.Validate(typeof(Widget), this);
        // Additional constructor implementation omitted
   }
     // Implements Dispose to dispose the license
   protected override void Dispose(bool Disposing)
   {
      if (myLicense != null)
      {
         myLicense.Dispose();
         myLicense = null;
      }
   }
}
```

7.3.4. You can host your Windows Form control in Internet Explorer by creating an <OBJECT> tag. The <OBJECT> tag specifies which control to load in the *classid* property. The Control must either be installed to the Global Assembly Cache, or it must reside in the same virtual directory as the HTML page it is declared in.

```
//The classid is divided in
//<path to the DLL with the control>#<NameSpace.Controlname>
<OBJECT id="myControl" classid="http:ControlLibrary1.dll#ControlLibrary1.myControl
VIEWASTEXT>
</OBJECT>
```

8. Advanced .NET Framework Topics

8.1. Implementing Print Functionality

8.1.1. PrintDocument class. An instance of this class represent a printed document and it is configured for the default printer. It presents the following properties:

8.1.1.1. PrinterSettings that specifies the settings for the printer

8.1.1.2. DefaultPageSettings that specifies the default page properties

8.1.1.3. PrintController that describes how each page is guided through the printing process

8.1.2. PrintDocument.Print method. Utililized to print a document, it fires the PrintPage event. You provide the logic for rendering your document to the printer in the PrintPage event handler.

8.1.2.1. PrintPageEventArgs object contains all of the information and functionality needed to render output to the printer. It provides a Graphics object to be used to print in the same way that when use it in forms, including positioning using coordinated in pixels.

8.1.2.1.1. For multiline documents, implement logic to handle paging. Set the property HasMorePages = true to fire the PrintPage event again.

PrintPageEventsArgs object properties

Cancel	Set to True to cancel the print Job
Graphics	Used to render content to the printed page
HasMorePages	True when multiple pages have to be printed
MarginBounds	Get the rectangle object that represents the portion within the margins
PageBounds	Get the rectangle object that represents the total page area
PageSettings	Gets or sets the PageSettings object for the current page

```
Using System.Drawing.Printing;
    .
    .
Private PrintDocument myPrintDocument;
    .
    .
myPrintDocument = new PrintDocument();
    .
    .
myPrintDocument.PrintPage += PrintPageEvenHandler(myPrintMethod);
    .
    .
//Printing a graphics in two pages
bool FirstPagePrinted = false;

// This method must handle the PrintPage event
public void myPrintMethod(object sender, System.Drawing.Printing.PrintPageEventArgs e)
{
   if (FirstPagePrinted == false)
   {
      FirstPagePrinted = true;
      e.HasMorePages = true; //Control printed pages
      e.Graphics.DrawEllipse(Pens.Black, new Rectangle(0, 0,
                                              e.PageBounds.Width, e.PageBounds.Height * 2));
   }
   else
   {
      e.HasMorePages = false;
      FirstPagePrinted = false;
      //Continue drawing the elipse from the last point
      e.Graphics.DrawEllipse(Pens.Black, new Rectangle(0,
                        -(e.PageBounds.Height), e.PageBounds.Width, e.PageBounds.Height * 2));
   }
}

myPrintDocument.Print();

  //When printing text calculate line spacing, lines per page and position of each line
int ArrayCounter = 0;
Font myfont = new Font("Batang", 36, FontStyle.Regular, GraphicsUnit.Pixel);
  // This method handles a PrintDocument.PrintPage event
private void PrintStrings(object sender, PrintPageEventArgs e)
{
     // Variables to keep track of spacing and paging
   float LeftMargin = e.MarginBounds.Left;
   float TopMargin  = e.MarginBounds.Top;
   float MyLines    = 0;
```

```
   float YPosition  = 0;
   int Counter      = 0;
   string CurrentLine;
     // Lines per page = margin height / font height
   MyLines = e.MarginBounds.Height / myFont.GetHeight(e.Graphics);
     // Prints each line, stop at the end of a page
   while (Counter < MyLines && ArrayCounter <= myStrings.GetUpperBound(0))
   {
      CurrentLine = myStrings[ArrayCounter];
      YPosition = TopMargin + Counter * myFont.GetHeight(e.Graphics);
      e.Graphics.DrawString(CurrentLine, myFont, Brushes.Black,
                                                 LeftMargin, YPosition, new StringFormat());
      Counter ++;
      ArrayCounter ++;
   }

     // If more lines exist, print another page.
   if (!(ArrayCounter == myStrings.GetUpperBound(0)))
      e.HasMorePages = true;
   else
      e.HasMorePages = false;
}
```

8.1.3. Color printers print in color by default. Change the *DefaultPageSettings.Color* property to False to force black and white. To check whether you have a color printer check for True in the property *PrinterSettings.SupportColor*.

8.1.4. The PrintPreviewControl control allows you to preview your print documents before printing them. The PrintPreviewDialog box incorporates the most commonly used PrintPreview features into an easy to use form. Add an instance of PrintPreviewControl by dragging it to your form.

```
//For PrintPreview Dialog
using System.Windows.Forms
       .
       .
private PrintPreviewDialog myDialog
       .
       .
myDialog = new PrintPreviewDialog();
myDialog.Document = myPrintDocument; //Associate the PrintDocument instance
       .
       .
  //Display the dialog
myDialog.Show();
myDialog.ShowDialog();

//For Print preview control (Using same namespace)
//The control has to be place in the form by setting position and size properties
PrintPreviewControl myPrintPreview
       .
       .
myPrintPreview = new PrintPreviewControl()
  //Associate the PrintDocument instance to preview printing
myPrintPreview.Document = myPrintDocument;
  //Update the preview by calling the InvalidatePreview method
myPrintPreview.InvalidatePreview();
```

8.1.5. The PrintDialog and the PageSetupDialog boxes allow users to configure printer and page properties at run time.

- In both cases use the Document property to set the PrintDocument object to manage and use the Show or ShowDialog methods to display them.
- PageSetupDialog updates the *PrintDocument.PrinterSettings* property.
- You can also configure individual page properties by changing the *PrintPageEventArgs.PageSettings* property during printing. Those changes apply only to the current page being print.

```
// This line is excerpted from a PrintPage event handler. The variable
// e represents the PrintPageEventArgs.
e.PageSettings.Landscape = true;
```

8.2. Accessing and Invoking Components

8.2.1. Using components.

You can use .NET assemblies or COM type libraries in your application by creating a reference to the type library and instantiating the relevant component. You can use ActiveX controls in the same way. Additionally, you can add them to the Toolbox.

8.2.2. Web Services

You can access a Web Service by adding a Web reference to your application. Once the Web reference is added, you can declare and instantiate the Web Service in code. This will create a wrapper class that exposes the methods of the Web Service. You can search the web for webServices by using the Universal Description Discovery Integration (UDDI) directory.

8.2.2.1. Synchronous calls to Web Service methods can be made like any other method call. Application execution will pause until the result from a synchronous call is returned. Those calls are made through a proxy class that represents the Web service with all its methods.

```
  //instantiating a Web service
WebService1 myService = new WebService1;
  //Calling a method in the WebService
myService.myMethod();
```

8.2.2.2. Asynchronous calls. For every method found on a Web Service, there are two additional methods for use asynchronously. Their namee are the name of the Web method prepended with Begin and End. Asynchronous calls are made in two parts. The Begin method requires a delegate to a method that receives an IAsyncResult as a parameter. The End method of an asynchronous method call uses the IAsyncResult returned by the Begin method as a parameter to retrieve the data it returns.

```
//Here MyMethod is a method on a Web Service called WebService1, and returns a string.
public class AsyncDemo
{
   WebService1 myService;
   public void CallMethodAsynchronously()
   {
      myService = new WebService1();
        // Creates the AsyncCallback delegate that will specify the callback method
      System.AsyncCallback myCallBack = new System.AsyncCallback(CallBack);
        // The object is required by the method call but is not used in this example.
      myService.BeginMyMethod(myCallBack, new object());
   }
   public void CallBack(IAsyncResult e)
   {
     string myString;
       //Call the 'End' method to retrieve the data using IAsyncResult as the parameter
     myString = myService.EndMyMethod(e);
   }
}

//Here the retrieve is made in the same method without using the
//CallBack method set in the delegate
public void AsyncDemo()
{
   WebService1 myService = new WebService1();
   IAsyncResult Async;
   string myString;
     // Creates the delegate to the callback method. This delegate is
     // required by the method call but will not be used to retrieve the data.
   System.AsyncCallback myCallBack = new System.AsyncCallback(SomeMethod);
   Async = myService.BeginMyMethod(myCallBack, new object());
     // Do some processor-intensive stuff here. If the call hasn't yet returned,
     //application execution will pause here until it does.
   myString = myService.EndMyMethod(Async);
}
```

8.2.3. External Functions.

You can declare external functions using the static and extern keywords. You must specify the name of the library that contains the function with the DllImportAttribute attribute. The name and signature of the function must match the name and signature of the external function exactly.

```
[System.Runtime.InteropServices.DllImport("kernel32")]
private static extern int Beep(int dwFreq, int dwDuration);
```

8.3. Implementing Accessibility.

8.3.1. When designing for accessibility, keep the following principles in mind:

- Flexibility
- Choice of input methods
- Choice of output methods

- Consistency
- Compatibility with accessibility aids

8.3.2. The Certified for Windows logo program has the following accessibility requirements for applications:

- Support standard system settings
- Be compatible with High Contrast mode
- Provide documented keyboard access for all UI features
- Provide notification of the focus location
- Convey no information by sound alone

8.3.3. Windows Forms controls expose the following accessibility-related properties:

- AccessibleDescription
- AccessibleName
- AccessibleRole
- AccessibilityObject
- AccessibleDefaultActionDescription

8.4. Implementing Help in Your Application

8.4.1. Help class. It provides static methods that allow you to display help for your application.

Use the Help.ShowHelp method to display help

```
using System.Windows.Forms;
// Display a help file for a particular control.
  // Specify the control and the path that can be an URL
Help.ShowHelp(MyForm, @"C:\myHelpFile.htm");
  // You can specify a HelpNavigator parameter giving the element to display
  // (TableOfContents, Find, Index, or Topic).
  //You can also specify a keyword to search for, as follows:
Help.ShowHelp(MyForm, @"C:\myHelpFile.htm", "HelpMenu");
  //To display the index of the help file use:
Help.ShowHelpIndex(MyForm, @"C:\myHelpFile.htm");
```

8.4.2. HelpProvider. component it is an extender provider that maintain properties for each control in the form.

8.4.2.1. To use it either set a HelpString for each control in the form or specify a HelpNameSpace with a URL to a help file.

```
Using System.Windows.Forms
HelpProvider myHelpProvider;
.
.
myHelpProvider = new HelpProvider();
.
.
HelpProvider.HelpNameSpace = @"c:\Helphere.html";
.
.
myHelpProvider.SetHelpString(myButton,"This is a help string");
```

8.4.2.2. The Help Provider has the following properties:

1. HelpString.
2. HelpKeyWorkd
3. HelpNavigator. Possible values are:
   - TableOfContents. Displays the table of contents page.
   - Find. Displays the search page.
   - Index. Displays the index.
   - Topic. Displays a help topic.
   - AssociatedIndex. Displays the index for a specified topic.
   - KeywordIndex. Displays a keyword-search-based result.

8.4.2.3. If the HelpNameSpace is not set, the HelpString will be displayed in a pop-up dialog box. If the HelpNameSpace is specified, the appropriate help file will be displayed instead.

8.5. Globalization and Localization

8.5.1. Globalization refers to formatting data appropriately based on locale. Localization refers to displaying the appropriate set of data based on locale.

8.5.2. .NET framework uses a culture code to specify languages (neutral culture), or language and region (specific culture). This code is formed by 2 letters to represent the language and sometimes plus other 2 letters to represent the language. You use this code to change the current culture in the application, and it is done with a new instance of the *CultureInfo* class. The class provide information about calendar, currencies, date format and more.

```
  //Setting the current culture to French-Canadian
 System.Threading.Thread.CurrentThread.CurrentCulture = new
                                                System.Globalization.CultureInfo("fr-CA");
  //Also you can retrieve the CultureInfo class of the application
  // This example assumes using System.GlobalizationCultureInfo
myCurrentCulture myCurrentCulture = CultureInfo.CurrentCulture;
```

8.5.3. You can implement globalization in your applications by using formatting for your data instead of hard coding the format. Formatting will update when the current culture changes.

```
  //Hardcode data won't be formated using the settings of the CUltureInfo class
Label1.Text = "$500.00";
  //But formated data will be formated to the according the culture
Label1.Text = (500).ToString("C");
```

8.5.4. Localization is implemented by providing multiple resource files for localized UI elements. These resource files are created automatically during Visual studio design for forms when the Localizable property is set to true and the Language property is changed. The CurrentCulture setting determines the kinds of formatting that will be applied to formatted data, whereas the CurrentUICulture determines the resources that will be loaded at run time for localized forms.

8.5.5. A particular culture is represented by an instance of *CultureInfo*. By setting *System.Threading.Thread.CurrentThread.CurrentCulture* to a new *CultureInfo* instance, you set the current culture. You can set the current UI thread by setting *System.Threading.Thread.CurrentThread.CurrentUICulture*.

```
  //You should set UI culture information in the constructor of the form
  //Retrieving the current UI culture
  // This example assumes using System.Globalization
CultureInfo myCurrentCulture;
myCurrentCulture = CultureInfo.CurrentUICulture;
  //Setting the UI culture to Thailand
System.Threading.Thread.CurrentThread.CurrentUICulture = new
                                                System.Globalization.CultureInfo("th-TH");
```

8.5.6. To validate non-latin input use *Char.IsDigit*, *Char.IsLetter* and similar methods. They will return appropriate values no matter what input character set is used.

8.5.7. You can create custom culture settings by setting individual members of a *CultureInfo* instance.

To do it customize the following members of the *CultureInfo* class:

DateTimeFormat	Calendar, date and time formatting
NumberFormat	Currency and percentaje formatting)
TextInfo	Text formatting, code page and list separator symbol

```
//Set culture to japanese but use Dollas currency ($)
// This example assumes using System.Globalization and using System.Threading
  CultureInfo modJPCulture = new CultureInfo("jp-JN");
  modJPCulture.NumberFormat.CurrencySymbol = "$";
  Thread.CurrentThread.CurrentCulture = modJPCulture;
```

8.5.8. To set Right-to-Left screens set the *RightToLeft* property to true. Every screen will be a mirror imagen of the original excepting the text displayed, which manually needs to be formatted.

8.5.9. You can convert data from code page encodings to Unicode, or any other encodings by using the *Encoding.Convert*. method. You can obtain an array of bytes from a string or char array by calling the *Encoding.GetBytes* method. Likewise, you can encode an array of bytes to an array of chars by calling the *Encoding.GetChars* method.

```
//Comverting data encode by vode 932 (Japanese) to data encoded in Unicode
// This example assumes using System.Text. The data to be converted
// is represented as a byte array by the variable name myData
byte[] tgtData;
Encoding srcEncoding;
UnicodeEncoding tgtEncoding = new UnicodeEncoding();
srcEncoding = Encoding.GetEncoding(932);    //static method returning a class instance
tgtData = Encoding.Convert(srcEncoding, tgtEncoding, myData);
//
  // tgtData now contains an array of bytes that represents the Unicode
  // encoding of the byte array in myData.
```

Helping extra code

```
  //To convert legacy data to an array of bytes in preparation for this conversion
  // This example assumes Imports System.Text. The data in the legacy
  // format is contained in a variable called myString
Encoding myEncoding;
myEncoding = Encoding.GetEncoding(932);
Byte[] myBytes;
myBytes = myEncoding.GetBytes(myString);
  //Likewise, you can use the GetChars method to convert
  //an array of bytes to an array of chars.
  // This example assumes Imports System.Text. The data is contained
  // in an array of bytes called myBytes.
UnicodeEncoding myEncoding = new UnicodeEncoding();
char[] myChars;
myChars = myEncoding.GetChars(myBytes);
```

9. Assemblies, Configuration, and Security

9.1. Assemblies and Resources

9.1.1. Assemblies are the building blocks of applications. they enablle code reuse, version control, security, and deployment. The identity of the assembly is contained in the AssemblyInfo.cs project file.

Parts of a Assemby"

1. The assembly manifest, or metadata
   i. Identity <name, version, locale, signature>
   ii. Type and Resources
   iii. Files <files inside with dependencies>
   iv. Security Permissions
2. The type metadata
3. The intermediate language code
4. Resource files

9.1.2. Resources files

9.1.2.1. Types of resource files:

- binary (.resources)
- XML (.resx).

Both can created using the ResEditor and then added to the project. After compilation they are embeded into the assembly.

9.1.2.2. Resource-only assemblies

They are created by opening an empty project and adding only resource files to it. The resulting DLL will contain resources that can be accessed from other assemblies, thus you avoid recompiling the application when the resources change.

9.1.2.3. Resource files for different cultures.

They are identified by using the language to their name. MyResources.resx => MyResources.**de**.resx (for german version). They will automatically be compiled into satellite assemblies, being automatically loaded based on the CurrentUICulture setting of the thread. After adding them to the project Visual Studio will place them in the proper directory structure.

9.1.3. ResourceManager class.

It is utilized to retrieve resources from a particular resource file. If satellite assemblies exist for a particular culture, the *ResourceManager* will automatically retrieve the appropriate resource when the CurrentUICulture is set to that culture.

Each instance of the ResourceManager is associated with a particular resource file embedded within a particular assembly.

The parameters are the namespace plus the resource file without extension and the assembly where it is contented.

Culture specific notations are not used (.de. for example).

```
  // Creating a ResourceManager to access resource n the embedded file myResources.resx in
  // the namespace myNamespace and the same assembly as the current object.
  //This example assumes Imports System.Resources
ResourceManager myManager = new ResourceManager ("myNamespace.myResources",
                                                 this.GetType().Assembly);
  //Accessing resources from another assembly this example assumes Imports System.Resources
System.Reflection.Assembly myResources;
  // The Assembly.Load method requires the name of the assembly as a parameter
myResources = System.Reflection.Assembly.Load("ResourceAssembly");
ResourceManager myManager = new ResourceManager("ResourceAssembly.Resources", myResources);
  //Retrieving resources using the ResourceManager
  // Assumes the existence of a ResourceManager named myManager
```

```
System.Drawing.Image myImage;
myImage = (System.Drawing.Image)myManager.GetObject("ImageResource");
```

9.1.4. Shared assemblies

9.1.4.1. Private Assembilies. Most assemblies are private and they can only be accessed by the application it is associated with. When a reference is made in a project to a nonshared assembly, a copy of that assembly is made and added to the project folder.

9.1.4.2. Shared assemblies can be accessed by multiple programs at once and they must be installed to the Global Assembly Cache. To install an assembly to the Global Assembly Cache, you must first sign it with a strong name. You can then use the Global Assembly Cache utility (gacutil.exe) to install the assembly to the Global Assembly Cache.

- To make a strong name do the following
  - Use **sn.exe** to create a file with the private/public key pair
    sn –k myKey.snk
  - Verify that the version information has been set for the assembly
  - Add the following attribute to the assembly file:
    [assembly: AssemblyKeyFile("..\\..\\myKey.snk")]
  - To install the assembly in the Global Assembly Cache use the gacutil.exe program:
    `gacutil /i myAssembly.dll`

9.2. Configuring and Optimizing Your Application

9.2.1. Configuration Files

- They are XML files located in the same directory than the application assembly that it configures and with the following naming convention:
  <Applicationname>.<extension>.config example: myApplication.exe.config
- They contain information that allows you to apply different configurations to your application without recompiling. You can configure applications by making changes in the configuration with a text editor, saving, and then restarting the program.
- The format of the configuration files is:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
      <!—configured elements go here —>
</configuration>
```

and some of the schema elements that it can contain are:

<startup>	Specifies the CLR in <requiredRuntime>
<runtime>	Info about assembly building and garbage collection
<system.runtime.remoting>	config. of channels and remote objects
<system.net>	For Internet appl.
<mscorlib>	Cryptography configuaration in <cryptographySettings>
<configSections>	Custom Cong. Settings
<system.diagnostics>	Cong. of Trace and Debug classes

9.2.2. Dynamic properties

- They are useful to describe external resources that are expected to change in the lifetime of an application.
- You can configure dynamic applications for UI elements in the designer using the Properties window . When they are set, the key and the value of that property are automatically written to the .config file
  <add key="Button1.Text" value="Button1" />
- You can only configure properties that are represented as strings or types that can be explicitly converted from a string. For example for booleans use "true" or "false".

9.2.2.1. AppSettingsReader.

- You can use the *System.Configuration.AppSettingsReader* class to read data from the .config file manually.
- The *AppSettingsReader.GetValue* method uses a key to retrieve values stored in <add> elements in the configuration file. and the type of the object to be retrieved. Also you have to cast the retrieved object to the proper type

```
  // Creates the AppSettingsReader object
System.Configuration.AppSettingsReader myReader = new
                                          System.Configuration.AppSettingsReader();
  // Creates a new Widget
Widget myWidget = new Widget();
```

```
  // Retrieves the dynamic property. DynamicWidget.Text is the key, and it is returned
  // as a string. It is converted from object to string explicitly.
myWidget.Text = myReader.GetValue("DynamicWidget.Text",typeof(System.String)).ToString();
```

And the configuration file could be

```
<appSettings>
   <add key="Widget.Visible" value="True" />
   <add key="Widget.Text" value="I love my Widget!" />
</appSettings>
```

9.2.3. Optimazing

9.2.3.1. You can use the perfmon.exe utility and Trace statements to monitor your application's performance. Use good coding practice and successive rounds of performance monitoring and code tuning to optimize your application.

9.2.3.2. You can enable compiler optimizations by selecting Optimize code (for Visual C#) on your application's property pages. This should only be done for release builds.

9.3. Securing Your Application

There are two basic types of code security: *role-based* security, which authenticates users and roles, and code-access security, which protects system resources from unauthorized calls

9.3.1. Role base security

- Role-based authorization verifies the role and/or identity of the current *Principal* object.
- The *Principal* object represents authenticated users and contains the users identity and role.

9.3.1.1. Using Windows built-in authentication.

9.3.1.1.1. To use the built-in windows security make use of the *WindowsPrincipal* object:

```
// This example assumes using System.Security.Principal
AppDomain.CurrentDomain.SetPrincipalPolicy (PrincipalPolicy.WindowsPrincipal);
```

9.3.1.1.2. The *WindowsIdentity* object represents the current user and it can extracted from the *WindowsPrincipal* object by using its *Identity* property:

```
WindowsPrincipal myPrincipal;
// Gets a reference to the current WindowsPrincipal
myPrincipal = (WindowsPrincipal) System.Threading.Thread.CurrentPrincipal;
WindowsIdentity myIdentity = (WindowsIdentity)myPrincipal.Identity;
MessageBox.Show(myIdentity.Name); // Displays the username
```

9.3.1.2. Imperative Role-Based security

Here you use a *PrincipalPermission* object that specifies authorized identity and roles, and uses the *Demand* method to verify that the current Principal objects match the authorized name and role. Example:

```
// Supply name and role as strings
PrincipalPermission myPermission = new PrincipalPermission("Megan","Manager");
// Demands that the CurrentPrincipal be named Megan in the role of Manager
myPermission.Demand();
```

9.3.1.3. Declarative Role-Based Security

Every permission object has a corresponding attribute. Those *Permission* attributes are attached to members to protect. Their constructor require a *SecurityAction* that indicates the action the permission should take (usually Demand).

```
[PrincipalPermission(SecurityAction.Demand, Name="Joe", Role="Clerk")]
public void myMethod()
{
   // Method implementation omitted
}
```

9.3.2. Code Access Security

It is used to protect code, but also can be used to communicate security requirements to the system administrator. Here permissions represent system resources and control access to those resources (*FileIOPermission* object, for example). Most of these permission are located in the *System.Security* namespace, Examples are:

DirectoryServicesPermission
EnvironmentPermission
EventLogPermission
FileDialogPermission
FileIOPermission
OleDbPermission

PrintingPermission
ReflectionPermission
RegistryPermission
SecurityPermission
SQLClientPermission
UIPermission

9.3.2.1. Code access permission inherit from *CodeAccessPermission*, and some of the most common methods are:

Method	Description
Assert	Asserts that the code calling this method can access the resource represented by the permission even if callers higher in the call stack do not have that permission.
Demand	Requires that all callers higher in the call stack have permission to access the resource represented by this permission.
Deny	Denies code that calls this method permission to access the resource represented by this permission.
PermitOnly	Denies code that calls this method permission to access the resource represented by this permission except for the subset of that resource specified by this permission.
RevertAll	Removes all previous Assert, Deny, and PermitOnly overrides.
RevertAssert	Removes all previous Assert overrides.
RevertDeny	Removes all previous Deny overrides.
RevertPermitOnly	Removes all previous PermitOnly overrides.

9.3.2.2. Imperative Code Access Security

Enforces security at run time. Permisions can be created to provide full, none or some access to resources

```
//***Represents completely restricted access to UI resources
UIPermission anotherPermission = new UIPermission(PermissionState.None);
// Creates a permission object that represents unrestricted access to the file system

//***Using the Demand method
FileIOPermission myPermission = new FileIOPermission(PermissionState.Unrestricted);
// Demands all callers to this code have unrestricted access permissionto the file system
myPermission.Demand();

//***Denies callers to the code permission, even if they have been granted by the CLR
// Creates a permission object that represents unrestricted access to the file system
FileIOPermission myPermission = new FileIOPermission(PermissionState.Unrestricted);
// Denies access to the file system to this method
myPermission.Deny();

//***Using the PermitOnly method
// Creates a permission object that represents access to a specific file
FileIOPermission myPermission = new FileIOPermission(FileIOPermissionAccess.Write,
                                                     "C:\\myFile.txt")
;
// Permits only access to this file, and denies all other permission
myPermission.PermitOnly();
```

9.3.2.2.1. The *Assert* method causes any *Demand* stack walks to cease checking for permission from callers higher in the stack. Thus, if an untrusted assembly calls a method containing an Assert call, which then attempts to call a method protected by a Demand, the Demand will be satisfied by the Assert .

```
// Creates a permission object that represents unrestricted access to the file system
FileIOPermission myPermission = new FileIOPermission(PermissionState.Unrestricted);
```

```
// Asserts permission to access the file system
myPermission.Assert();
```

9.3.2.2.2. Revert methods are static and affect all the objects of the type.

```
// Reverts all ReflectionPermission overrides
ReflectionPermission.RevertAll();
// Reverts EnvironmentPermission Deny calls
EnvironmentPermission.RevertDeny();
// Reverts FileIOPermission Assert calls
FileIOPermission.RevertAssert();
// Reverts MessageQueuePermission PermitOnly calls
MessageQueuePermission.RevertPermitOnly();
```

9.3.3. Declarative Code Access Security.

Implemented using attributes, here you have to specify the security action represented by the attribute.

```
//Example. Denying FileIOPermission to a class using declarative security
[FileIOPermission(SecurityAction.Deny)]
public class aClass
{
   // Class implementation omitted
}
```

9.3.3.1. The SecurityAction.Demand, SecurityAction.Deny, SecurityAction.Assert, and SecurityAction.PermitOnly flags correspond to the Demand, Deny, Assert, and PermitOnly methods of the relevant permission, respectively. SecurityAction.LinkDemand requires the immediate caller to this class to have granted the appropiate permission.

9.3.3.2. To set permission attributes to the whole assembly use the following SecurityFlags:

SecurityAction.RequestMinimum	Requestes a permission the the CLR, if the security policy doesn't grant it, the assembly will not execute.
SecurityAction.RequesOptional.	Here the assembly still will run even if the permission is not granted.
SecurityAction.RequestRefuse.	Requests that the assembly be denied the specific permission.

```
[assembly: FileIOPermission(SecurityAction.RequestMinimum)]
```

9.3.3.3. You can set any properties of the permission attribute upon initialization. The following example demonstrates how to create an Assert for permission to access a single file using declarative security:

```
[FileIOPermission(SecurityAction.Assert, Write="C:\\myFile.txt")]
public void WriteFile()
{
   // Method implementation omitted
}
```

10. Deploying your Application

10.1. Planning the Deployment of your Project

10.1.1. To deploy a .NET applications by using XCOPY you have to have the NET Framework installed and the application doesn't have to have dependencies on shared assemblies or resources.

10.1.2. You can use Visual Studio .NET to create Windows Installer setup projects and merge modules. A setup project must be executed on a computer that has Windows Installer 1.5 installed and contains all of the content and information needed to install an application to a client computer. A merge module is used to package DLL files and cannot be installed by itself—it must be merged with a setup project.

10.1.3. The Setup Project wizard allows you to specify the content to add to your setup project and choose the kind of setup project to create. You can add other files needed such as HTML help and readme files. Also Visual Studio detects any dependencies and adds the to the "Detected Dependencies" folder.

10.1.4. You can configure the *Build* properties of your setup project by using the Setup Property Pages. You can configure how the files are packaged, compression settings, whether to create a bootstrapper, the output file directory, and Authenticode settings. The minimum file created is <projectName>.msi for Windows Installer applications. Other possible file would be <projectName>.msm for Windows Installer merge modules.

10.1.5. The default packing of files is in the setup file itself, but also it is possible to pack them in CAB files of a specific size. Another option is in loose, uncompressed file; the files just are copied to the same directory as the MSI file.

10.1.6. A bootstrapping application is used to install Windows Installer 1.5 to the target machine before your application is installed, and it can be installed in Windows 95/98/NT/2000. Windows XP already has it installed. To make the application downloadable from the Web choose instead the Web Bootstrapper option.

10.1.7. You have to chose the setup project in the Solution Explorer and build it. Once the project is built, you can distribute it in a variety of ways. You should configure the Build properties of your setup project to be optimized for your distribution plan.

10.1.8. Distribution via Removable Media. Place the MSI file in the first disk, and then copy each CAB file to a separate disk.

- Distribuition via a Network Share
- Distribuition via a Network Share Using Administrative Installation.
- Distribuition via the World Wide Web.

.

10.1.9. If a BootStrapper was included it will be installed when clicking the setup.exe file. To no install it make click on the MSI file. In anyway, always the MSI is processed

10.2. Configuring your SetUp Project

10.2.1. Setup applications are highly configurable. You can set setup properties that provide information about the origin of your application and behavior of your application at design time in the Properties window. You can provide icons for the "Add/Remove" dialogs, Author, Description, Localization, ProductName and others. Also you can set the "DetectNewerInstall" to abort if a newer version is already installed, "RemovePreviousVersion" to remove it, and "Version" to determine versioning.

10.2.2. Use the Register property in the properties window to register a COM component or Font at install time.

10.2.3. Use the File System Editor to edit the file system on the target computer, add shortcuts and install an Assembly to the Global Assembly Cache

10.2.4. Use the Registry Editor to write registry entries to the target computer.

10.2.5. Use the File Types Editor to create file associations on the target computer. You have to receive the path to the file (and optionally other paremeters) as a parameter of the "main" method. You have to modify the Main method to receive them.

```
static void Main(string[] args)
{
   if(args.Length!=0)
   {
      // Assumes that the application's Open command takes a
      // string that contains the path of the file to open. The
      // Open method must also be static, as you cannot call an
      // instance method from a static method.
      Open(args[0]);
   }
   else
   {
      Application.Run(new Form1());
   }
}
```

10.2.6. Use the User Interface Editor to customize the installation user interface. The editor display two view displays: Install and Administrative Install (for administrators), each one of them shows the phases Start (gather info. Asks for directories), Progress (progress bar), and End (final info. About the installation). You can also add customizable dialog boxes to any phase.

10.2.7. Use the Custom Actions Editor to add a custom action. They are code that can be executed after the following Installer events: Install, Commit, Rollback, and Uninstall.

10.2.8. Use the Launch Conditions Editor to add a search and a launch condition. Those are condicions that the target computer has to met to procede with the installation. You can search for files, registry entries, and Windows Installer components.

10.2.9. Use Ngen.exe to create a native code image of your application or assembly. This utility installs a precompiled version to native code installing it in the Native Image Cache. The run time will execute the native imagen whenever the assembly is called.

```
  //For DLL assemblies, specify the name of the assembly.
  //For executables specify the path to the executable file
Ngen.exe myAssemblyNgen.exe C:\myApp.exe
```

10.2.10. Use Permview.exe to view the permissions granted to an assembly.

```
Permview.exe myAssembly.dll
Permview.exe myApp.exe
  //To get class and method leve permissions
Permview.exe /DECL myAssembly.dll
```

```
  //To write the output to a file
Permview.exe /OUTPUT myTextFile.text myAssembly.dll
```