

Coalesce

Contents

	Page
1. .Net Defined	2
2. The Visual Basic Language	6
3. Object Oriented Programming Basics	10
4. Classes and Objects	12
5. Data Types And Variables	24
6. Control Flow Statements	32
7. Arrays and String manipulation	36
8. Procedures	42
9. Exception Handling	44
10. Statements and Scope	46
11. Win Forms	48
12. More On Windows forms	59
13. MDI Applications	63
14. Controls	74
15. I/O Handling	109
16. Deploying Applications	120
17. Debugging	127

Coalesce

Chapter 1

.NET Defined

Before getting deeply into the subject we will first know how Businesses are related to Internet, what .NET means to them and what exactly .NET is built upon. As per the product documentation, from a Business perspective, there are three phases of the Internet. The First phase gets back to the early 1990's when Internet first came into general use and which brought a big revolution for Businesses. In the First phase of the Internet Businesses designed and launched their Website's and focused on the number of hits to know how many customers were visiting their site and interested in their products etc. The Second phase is what we are in right now and in this phase Businesses are generating revenue through Online Transactions. We are now moving into the Third phase of the Internet where profit is the main priority. The focus here is to Businesses effectively communicate with their customers and partners, who are geographically isolated, participate in Digital Economy and deliver a wide range of services. How can that be possible? The answer is, with .NET.

1.1 What is .NET ?

Many people reckon that it's Microsoft's way of controlling the Internet, which is false. .NET is Microsoft's strategy of software that provides services to people *any time, any place on any device*. An accurate definition of .NET is, it's an *XML Web Services platform* which allows us to build rich .NET applications, which allows users to interact with the Internet using wide range of *smart devices* (tablet devices, pocket PC's, web phones etc), which allows to build and integrate Web Services and which comes with many rich set of tools like *Visual Studio* to fully develop and build those applications.

1.2 What are Web Services?

Web Services are the applications that run on a Web Server and communicate with other applications. It uses a series of protocols to respond to different requests. The protocols on which Web Services are built are summarized below:

UDDI : Stands for *Universal Discovery and Description Integration*. It's said to be the Yellow Pages of Web Services which allows Businesses to search for other Businesses allowing them to search for the services it needs, know about the services and contact them.

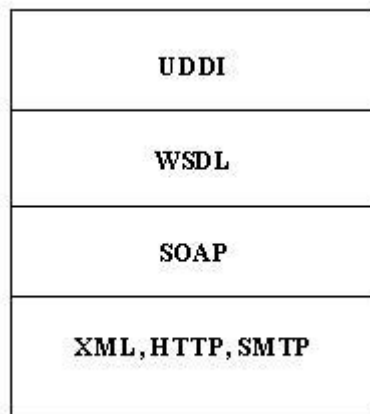
WSDL: Stands for *Web Services Description Language*, often called as *whiz-dull*. WSDL is an XML document that describes a set of SOAP messages and how those messages are exchanged.

Coalesce

SOAP: Stands for *Simple Object Access Protocol*. It's the communication protocol for Web Services.

XML, HTTP and SMTP: Stands for *Extensible Markup Language, Hyper Text Transfer Protocol and Simple Message Transfer Protocol* respectively. UDDI, WSDL and SOAP rely on these protocols for communication.

The image below shows the order of the protocols on which Web Services are built:



Example of a Web Services Application

Let's say a customer accesses a Website and buys something. The Web services of the business will communicate with the inventory system to see if there is enough stock to fulfill the order. If not, the system can communicate with the suppliers to find one or all of the parts that make up the order before filling the order. At all stages the customer will be kept informed via messages. The end result is a seamless system communicating and exchanging information easily regardless of the platform they are all running on. The business doesn't need to worry about going to the wrong supplier because it asks the Web service running on the supplier system what it does. And the business doesn't have to worry about the other system's methods of handling data because they communicate via SOAP and XML.

Real World Application

Microsoft's passport service is an example of .NET service. Passport is a Web-based service designed to make signing in to websites fast and easy. Passport enables participating sites to authenticate a user with a single set of sign-in credentials, eliminating the need for users to remember numerous passwords and sign-in names. You can use one name and password to sign in to all .NET Passport-participating sites and services. You can store personal information in your .NET Passport profile and, if you choose, automatically share that information when you sign in so that participating sites can provide you with personalized services. If you use Hotmail for your email needs then you should be very much familiar with the passport service.

To find out more about how Businesses are implementing Web Services and the advantages it is providing please visit Microsoft's Website and check out the case studies published.

1.3 What is .NET Built On?

Coalesce

.NET is built on the *Windows Server System* to take major advantage of the OS and which comes with a host of different servers which allows for building, deploying, managing and maintaining Web-based solutions. The Windows Server System is designed with performance as priority and it provides scalability, reliability, and manageability for the global, Web-enabled enterprise. The Windows Server System integrated software products are built for interoperability using open Web standards such as XML and SOAP.

Core Windows Server System Products include :

SQL Server 2000: This Database Server is Web enabled and is designed with priority for .NET based applications. It is scalable, easy to manage and has a native XML store.

Application Center 2000: This product manages Web Applications.

Commerce Server 2000: This powerful Server is designed for creating E-Commerce based applications.

Mobile Information Server: This Server provides real-time access for the mobile community. Now Outlook users can use their Pocket PC's to access all their Outlook data while they are moving.

Exchange Server 2000: This is a messaging system Server and allows applications on any device to access information and collaborate using XML.

BizTalk Server 2000: This is the first product created for .NET which is XML based and allows to build business process that integrate with other services in the organization or with other Businesses.

Internet Security and Acceleration Server 2000: This Server provides Security and Protection for machines. It is an integrated firewall and Web cache server built to make the Web-enabled enterprise safer, faster, and more manageable.

Host Integration Server 2000: This Server allows for the Integration of mainframe systems with .NET.

When developing real world projects if you don't know how to use the above mentioned Server's which are built for .NET based applications, do not panic. Your System Administrator is always there to help you.

1.4 .NET and XML

There is a lot of connection between XML and .NET. XML is the glue that holds .NET together. XML looks similar to HTML which is readable and text-based. XML is a method of putting structured data into a text file. XML is the specification for defining the structure of the document. Around this specification a whole family of optional modules are being developed. The reason why XML is linked so much to .NET is it's platform independent and is well supported on any environment. To move the data contained in an XML file around different organizations using different software on different platforms, it should be packed it into something. That something is a protocol like SOAP.

About SOAP

Coalesce

SOAP, the Simple Object Access Protocol, is a simple, lightweight protocol for exchanging information between peers in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelop that describes what is in the message and how it should be processed, a set of encoding rules and a convention for representing remote procedure calls and responses.

1.5 .NET vs Java

Many of us wonder what .NET has to do with Java. Is there any relation between them? Are they similar? and so on. I even hear some people say that .NET is Microsoft's answer to Java. I think every language has it's own pros and cons. Java is a very good programming language but requires us to write heaps of code to develop applications. When it comes to .NET, the Framework supports around 20 different programming languages which are better and focus only on the business logic leaving all other aspects to the Framework. Many applications were developed, tested and maintained to compare the differences between .NET and Java and the end result was a particular application developed using .NET requires less lines of code, less time to develop and lower deployment costs along with other important issues. Personally, I don't mean to say that Java is totally gone or .NET based applications are going to dominate the Internet but I think .NET definitely has an extra edge as it is packed with many features.

I hope the information above puts some light on the technology aspects behind .NET and helps you in getting started. I will publish more information about .NET technology, it's powerful features, and much more when I will rehash the site in future.

Coalesce

Chapter 2

The Visual Basic Language

Visual Basic, the name itself makes me feel that it is something special. In the History of Computing world no other product sold more copies than Visual Basic did. Such is the importance of that language which clearly states how widely it is used for developing applications. Visual Basic is very popular for it's friendly working (graphical) environment. Visual Basic .NET is an extension of Visual Basic programming language with many new features in it. The changes from VB to VB .NET are huge, ranging from the change in syntax of the language to the types of projects we can create now and the way we design applications. Visual Basic .NET was designed to take advantage of the .NET Framework base classes and runtime environment. It comes with power packed features that simplifies application development.

Briefly on some of the changes :

- ▶ Perhaps the biggest change from VB to VB .NET is, VB .NET is Object Oriented now. Being Object Oriented means that VB .NET now supports all the key OOP features like Inheritance, Polymorphism, Abstraction and Encapsulation. We can now create classes and objects, derive classes from other classes and so on. The major advantage of OOP is *code reusability*
- ▶ The Command1 Button is now Button1 and the TextBox as TextBox1 instead of Text1, in VB6
- ▶ Many new controls have been added to the toolbar to make application development more efficient
- ▶ VB .NET now adds Console Applications to it apart from Windows and Web Applications. Console applications are console oriented applications that run in the DOS version
- ▶ All the built in VB functionality now is encapsulated in a *Namespace* (collection of different classes) called *System*

- ▶ New keywords are added and old one's are either removed or renamed
- ▶ VB .NET is *strongly typed* which means that we need to declare all the variables by default before using them
- ▶ VB .NET now supports structured exception handling using *Try...Catch...Finally* syntax
- ▶ The syntax for procedures is changed. Get and Let are replaced by Get and Set
- ▶ Event handling procedures are now passed only two parameters
- ▶ The way we handle data with databases is changed as well. VB.NET now uses ADO .NET, a new data handling model to communicate with databases on local machines or on a network and also it makes handling of data on the Internet easy. All the data in ADO .NET is represented in XML format and is exchanged in the same format. Representing data in XML format allows us for sending large amounts of data on the Internet and it also reduces network traffic when communicating the with database

Coalesce

- ▶ VB .NET now supports Multithreading. A threaded application allows to do number of different things at once, running different execution threads allowing to use system resources
- ▶ Web Development is now an integral part of VB .NET making *Web Forms* and *Web Services* two major types of applications

Namespaces

A namespace is a *collection of different classes*. All VB applications are developed using classes from the .NET System namespace. The namespace with all the built-in VB functionality is the *System* namespace. All other namespaces are based on this System namespace.

Some of the namespaces:-

System: Includes essential classes and base classes for commonly used data types, events, exceptions and so on

System.Data: Includes classes which lets us handle data from data sources

System.Drawing: Gives access to drawing

System.IO: Includes classes for data access with Files

System.Net: Provides interface to protocols used on the internet

System.Security: Includes classes to support the structure of common language runtime security system

System.Threading: Includes classes and interfaces to support multithreaded applications

System.Web: Includes classes and interfaces that support browser-server communication

System.Windows.Forms: Includes classes for creating Windows based forms

System.XML: Includes classes for XML support

Assemblies

An assembly is the *building block* of a VB .NET application. An Assembly is a compiled and versioned collection of code and metadata that forms an atomic functional unit. Assemblies take the form of a dynamic link library (.dll) file or executable program file (.exe) but they differ as they contain the information found in a type library and the information about everything else needed to use an application or component. All the .NET programs are constructed from these Assemblies. Assemblies are made of two parts. First, the *manifest*, which is similar to a table of contents which gives the name and version of the assembly. Second, the *modules*, which are internal files of *IL code* which are ready to run. When programming, we don't directly deal with assemblies as the CLR and the .NET framework takes care of that behind the scenes. The assembly file is visible in the Solution Explorer window of the project.

An assembly includes:

- Information for each public class or type used in the assembly – information includes class or type names, the classes from which an individual class is derived, etc
- Information on all public methods in each class like the method name and return values(if any)
- Information on every public parameter for each method like the parameter's name and type
- Information on public enumerations including names and values
- Information on the assembly version (each assembly has a specific version number)

Coalesce

- Intermediate language code to execute
- Required resources such as pictures, assembly metadata

Visual Basic .NET (2003) provides the easiest, most productive language and tool for rapidly building Windows and Web applications. Visual Basic .NET comes with enhanced visual designers, increased application performance, and a powerful integrated development environment (IDE). It also supports creation of applications for wireless, Internet-enabled hand-held devices. The following are the features of Visual Basic .NET with .NET Framework 1.0 and Visual Basic .NET 2003 with .NET Framework 1.1. This also answers, why should I use Visual Basic .NET, what can I do with it?

Powerful Windows-based Applications

Visual Basic .NET comes with features such as a powerful new forms designer, an in-place menu editor, and automatic control anchoring and docking. Visual Basic .NET delivers new productivity features for building more robust applications easily and quickly. With an improved integrated development environment (IDE) and a significantly reduced startup time, Visual Basic .NET offers fast, automatic formatting of code as you type, improved IntelliSense, an enhanced object browser and XML designer, and much more.

Building Web-based Applications

With Visual Basic .NET we can create Web applications using the shared Web Forms Designer and the familiar "*drag and drop*" to build Web forms. You can double-click and write code to respond to events. Visual Basic .NET 2003 comes with an enhanced *HTML Editor* for working with complex Web pages. We can also Use IntelliSense technology and tag completion, or choose the *WYSIWYG editor* for visual authoring of interactive Web applications.

Simplified Deployment

With Visual Basic .NET we can build applications more rapidly and deploy and maintain them with efficiency. Visual Basic .NET 2003 and .NET Framework 1.1 makes "*DLL Hell*" a thing of the past. Side-by-side versioning enables multiple versions of the same component to live safely on the same machine so that applications can use a specific version of a component. *XCOPY-deployment* and *Web auto-download* of Windows-based applications combine the simplicity of Web page deployment and maintenance with the power of rich, responsive Windows-based applications.

Powerful, Flexible, Simplified Data Access

You can tackle any data access scenario easily with ADO.NET and ADO data access. The flexibility of ADO.NET enables data binding to any database, as well as classes, collections, and arrays, and provides true XML representation of data. Seamless access to ADO enables simple data access for connected data binding scenarios. Using ADO.NET, Visual Basic .NET can gain high-speed access to MS SQL Server, Oracle, DB2, Microsoft Access, and more.

Improved Coding

You can code faster and more effectively. A multitude of enhancements to the code editor, including enhanced IntelliSense, smart listing of code for greater readability and a background compiler for real-time notification of syntax errors transforms into a rapid application development (RAD) coding machine.

Coalesce

Direct Access to the Platform

Visual Basic developers can have full access to the capabilities available in .NET Framework 1.1. Developers can easily program system services including the event log, performance counters, and file system. The new Windows Service project template enables to build real Microsoft Windows NT Services. Programming against Windows Services and creating new Windows Services is not available in Visual Basic .NET Standard, it requires Visual Studio 2003 Professional, or higher.

Full Object-Oriented Constructs

You can create reusable, enterprise-class code using full object-oriented constructs. Language features include full implementation inheritance, encapsulation, and polymorphism. Structured exception handling provides a global error handler and eliminates spaghetti code.

XML Web Services

XML Web services enable you to call components running on any platform using open Internet protocols. Working with XML Web services is easier where enhancements simplify the discovery and consumption of XML Web services that are located within any firewall. XML Web services can be built as easily as you would build any class in Visual Basic 6.0. The XML Web service project template builds all underlying Web service infrastructure.

Mobile Applications

Visual Basic .NET 2003 and the .NET Framework 1.1 offer integrated support for developing mobile Web applications for more than 200 Internet-enabled mobile devices. These new features give developers a single, mobile Web interface and programming model to support a broad range of Web devices, including WML 1.1 for WAP—enabled cellular phones, compact HTML (cHTML) for i-Mode phones, and HTML for Pocket PC, handheld devices, and pagers. Please note, Pocket PC programming is not available in Visual Basic .NET Standard, it requires Visual Studio 2003 Professional, or higher.

COM Interoperability

You can maintain your existing code without the need to recode. COM interoperability enables you to leverage your existing code assets and offers seamless bi-directional communication between Visual Basic 6.0 and Visual Basic .NET applications.

Reuse Existing Investments

You can reuse all your existing ActiveX Controls. Windows Forms in Visual Basic .NET 2003 provide a robust container for existing ActiveX controls. In addition, full support for existing ADO code and data binding enable a smooth transition to Visual Basic .NET 2003.

Upgrade Wizard

You upgrade your code to receive all of the benefits of Visual Basic .NET 2003. The Visual Basic .NET Upgrade Wizard, available in Visual Basic .NET 2003 Standard Edition, and higher, upgrades

Coalesce

up to 95 percent of existing Visual Basic code and forms to Visual Basic .NET with new support for Web classes and UserControls.

Chapter 3

Object Oriented Programming Basics

Visual Basic was Object-Based, Visual Basic .NET is *Object-Oriented*, which means that it's a true Object-Oriented Programming Language. VB .NET supports all the key OOP features like *Polymorphism, Inheritance, Abstraction and Encapsulation*. It's worth having a brief overview of OOP before starting VB.NET.

3.1 Why Object Oriented approach?

A major factor in the invention of Object-Oriented approach is to remove some of the flaws encountered with the procedural approach. In OOP, data is treated as a critical element and does not allow it to flow freely. It bounds data closely to the functions that operate on it and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. A major advantage of OOP is code reusability.

Some important features of Object Oriented programming are as follows:

- Emphasis on data rather than procedure
- Programs are divided into Objects
- Data is hidden and cannot be accessed by external functions
- Objects can communicate with each other through functions
- New data and functions can be easily added whenever necessary
- Follows bottom-up approach

3.2 Concepts of OOP:

- Objects
- Classes
- Data Abstraction and Encapsulation
- Inheritance
- Polymorphism

3.2.1 Briefly on Concepts:

Coalesce

Objects

Objects are the basic run-time entities in an object-oriented system. Programming problem is analyzed in terms of objects and nature of communication between them. When a program is executed, objects interact with each other by sending messages. Different objects can also interact with each other without knowing the details of their data or code.

Classes

A *class* is a collection of objects of similar type. Once a class is defined, any number of objects can be created which belong to that class.

Data Abstraction and Encapsulation

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes.

Storing data and functions in a single unit (class) is *encapsulation*. Data cannot be accessible to the outside world and only those functions which are stored in the class can access it.

Inheritance

Inheritance is the process by which objects can acquire the properties of objects of other class. In OOP, *inheritance* provides reusability, like adding additional features to an existing class without modifying it. This is achieved by deriving a new class from the existing one. The new class will have combined features of both the classes.

Polymorphism

Polymorphism means the ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends on the data types used in the operation. Polymorphism is extensively used in implementing Inheritance.

3.2.2 Advantages of OOP

Object-Oriented Programming has the following advantages over conventional approaches:

- OOP provides a clear modular structure for programs which makes it good for defining abstract datatypes, where implementation details are hidden and the unit has a clearly defined interface.

- OOP makes it easy to maintain and modify existing code, as new objects can be created with small differences to existing ones.

- OOP provides a good framework for code libraries, where supplied software components can be easily adapted and modified by the programmer. This is particularly useful for developing graphical user interfaces.

3.3 OOP in Visual Basic

Coalesce

Visual Basic .NET is Object-Oriented. Everything we do in Visual Basic involves objects in some way or other and everything is based on the Object class. Controls, Forms, Modules etc are all types of classes. Visual Basic .NET comes with thousands of built-in classes which are ready to use. Let's take a closer look at Object Oriented Programming in Visual Basic. We will see how we can create classes, objects, how to inherit one class from other, what is polymorphism, how to implement interfaces and so on. We will work with Console Applications here as they are simple to code.

Chapter 4

Classes and Objects

4.1 Classes And Objects

Classes are *types* and Objects are *instances* of the Class. Classes and Objects are very much related to each other. Without objects you can't use a class. In Visual Basic we create a class with the *Class* statement and end it with *End Class*. The Syntax for a Class looks as follows:

```
Public Class Test
----Variables
----Methods
----Properties
----Events
End Class
```

The above syntax created a class named Test. To create an object of this class we use the *new* keyword and that looks like this: **Dim obj as new Test()** . The following code shows how to create a Class and access the class with an Object. Open a Console Application and place the following code in it.

```
Imports System.Console
Module Module1

Sub Main()
Dim obj As New Test()
'creating a Object obj for Test class
obj.disp()
'calling the disp method using obj
Read()
End Sub
```

Coalesce

```
End Module

Public Class Test
'creating a class named Test

Sub disp()
'a method named disp in the class

Write("Welcome to OOP")

End Sub

End Class
```

Output of above code is the image below.



Fields, Properties, Methods and Events

Fields, Properties, Methods, and Events are members of the class. They can be declared as Public, Private, Protected, Friend or Protected Friend.

Fields and Properties represent information that an object contains. *Fields* of a class are like variables and they can be read or set directly. For example, if you have an object named House, you can store the numbers of rooms in it in a field named Rooms. It looks like this:

```
Public Class House
Public Rooms as Integer
End Class
```

Properties are retrieved and set like fields, but are implemented using *Property Get* and *Property Set* procedures which provide more control on how values are set or returned.

Methods represent the object's built-in procedures. For example, a Class named Country may have methods named Area and Population. You define methods by adding procedures, Sub routines or

Coalesce

functions, to your class. For example, implementation of the Area and Population methods discussed above might look like this

```
Public Class Country
Public Sub Area()
Write("-----")
End Sub
Public Sub population()
Write("-----")
End Sub
End Class
```

Events allow objects to perform actions whenever a specific occurrence takes place. For example when we click a Button, a Click event occurs and we can handle that event in an event handler.

4.2 Inheritance

A key feature of OOP is reusability. It's always time saving and useful if we can reuse something that already exists rather than trying to create the same thing again and again. Reusing the class that is tested, debugged and used many times can save us time and effort of developing and testing it again. Once a class has been written and tested it can be used by other programs to suit the program's requirement. This is done by creating a new class from an existing class. The process of deriving a new class from an existing class is called Inheritance. The old class is called the *base class* and the new class is called *derived class*. The derived class inherits some or everything of the base class. In Visual Basic we use the *Inherits* keyword to inherit one class from other. The general form of deriving a new class from an existing class looks as follows:

```
Public Class One
---
---
End Class

Public Class Two
    Inherits One
---
---
End Class
```

Using Inheritance we can use the variables, methods, properties etc from base class and add more functionality to it in the derived class. The following code demonstrates the process of Inheritance in Visual Basic.

```
Imports System.Console
Module Module1

Sub Main()
Dim ss As New Two()
WriteLine(ss.sum())
Read()
End Sub
```

Coalesce

```
End Module

Public Class One
'base class
Public i As Integer = 10
Public j As Integer = 20

Public Function add() As Integer
Return i + j
End Function

End Class

Public Class Two
    Inherits One
'derived class. class two inherited from class one
Public K As Integer = 100

Public Function sum() As Integer
'using the variables, function from base class and adding more
functionality
Return i + j + K
End Function

End Class
```

Output of above code is sum of i, j, k as shown in the image below.



4.3 Polymorphism

Coalesce

Polymorphism is one of the crucial features of OOP. It means "*one name, multiple forms*". It is also called as *Overloading* which means the use of same thing for different purposes. Using Polymorphism we can create as many functions we want with one function name but with different argument list. The function performs different operations based on the argument list in the function call. The exact function to be invoked will be determined by checking the type and number of arguments in the function.

The following code demonstrates the implementation of Polymorphism.

```
Imports System.Console
Module Module1

Sub Main()
Dim two As New One()
WriteLine(two.add(10))
'calls the function with one argument
WriteLine(two.add(10, 20))
'calls the function with two arguments
WriteLine(two.add(10, 20, 30))
'calls the function with three arguments
Read()
End Sub

End Module

Public Class One
Public i, j, k As Integer

Public Function add(ByVal i As Integer) As Integer
'function with one argument
Return i
End Function

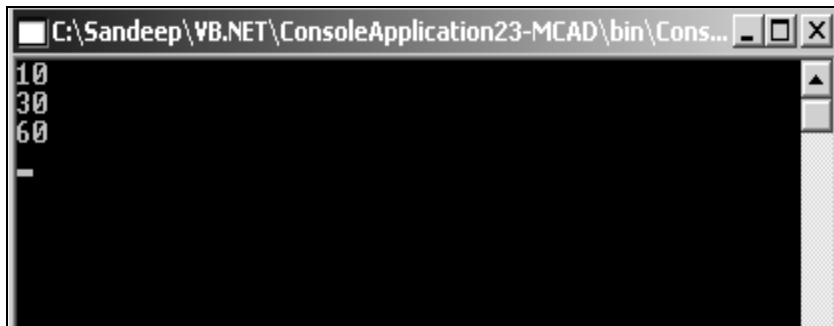
Public Function add(ByVal i As Integer, ByVal j As Integer) As Integer
'function with two arguments
Return i + j
End Function

Public Function add(ByVal i As Integer, ByVal j As Integer, ByVal k As Integer) As Integer
'function with three arguments
Return i + j + k
End Function
```


Coalesce

End Class

Output of the above code is shown in the image below.



4.4 Interfaces

Interfaces allow us to create definitions for component interaction. They also provide another way of implementing polymorphism. Through interfaces we specify methods that a component must implement without actually specifying how the method is implemented. We just specify the methods in an interface and leave it to the class to implement those methods. Visual Basic .NET does not support multiple inheritance directly but using interfaces we can achieve multiple inheritance. We use the *Interface* keyword to create an interface and *implements* keyword to implement the interface. Once you create an interface you need to implement all the methods specified in that interface. The following code demonstrates the use of interface.

```
Imports System.Console
Module Module1

    Sub Main()
        Dim OneObj As New One()
        Dim TwoObj As New Two()
        'creating objects of class One and Two
        OneObj.disp()
        OneObj.multiply()
        TwoObj.disp()
        TwoObj.multiply()
        'accessing the methods from classes as specified in the interface
    End Sub

End Module

Public Interface Test
    'creating an Interface named Test
    Sub disp()
    Function Multiply() As Double
    'specifying two methods in an interface
End Interface

Public Class One
    Implements Test
```

Coalesce

```
'implementing interface in class One

Public i As Double = 12
Public j As Double = 12.17

Sub disp() Implements Test.disp
'implementing the method specified in interface
WriteLine("sum of i+j is" & i + j)
Read()
End Sub

Public Function multiply() As Double Implements Test.Multiply
'implementing the method specified in interface
WriteLine(i * j)
Read()
End Function

End Class

Public Class Two
    Implements Test
'implementing the interface in class Two

Public a As Double = 20
Public b As Double = 32.17

Sub disp() Implements Test.disp
WriteLine("Welcome to Interfaces")
Read()
End Sub

Public Function multiply() As Double Implements Test.Multiply
WriteLine(a * b)
Read()
End Function

End Class
```

Output of above code is the image below.

Coalesce



4.5 Creating Namespaces

With Visual Basic .NET we can also create our own namespaces and import them just like all other namespaces. Creating our own namespaces are useful when we have large number of classes as they will help in organizing those classes. The parts of the namespace are as follows:

Name: A unique name that identifies the namespace. This is required.
Componenttypes: Elements that make up the namespace. This is optional.

Namespaces are always Public therefore the declaration of a namespace should not include access modifiers. The components within the namespace may have Public or Friend access. The default access type is Friend.

The following code shows how to create a namespace.

```
Namespace Test
' Declares a namespace named Test
'you can also have another namespace within this namespace

Class One
' Declares a class within Test
--class statements
--class statements
--class statements
End Class

End Namespace
```

Once you create the namespace you can't use the namespace straight away with the Imports keyword. To use the namespace which we created we need to create a dll (dynamic link library) file. To create a dll file for this namespace open Visual Studio .NET command prompt, go to the current directory and type the following command.

Coalesce

vbc /target:library Test.vb . Once you finish creating your dll, you can use it in projects by adding a reference to it. To add a reference to the dll you created, right-click on the Project name in the Solution Explorer window and select Add Reference. On the Add Reference window click the browse button, browse for the dll you created, select it and click OK. That adds a reference to the dll and you can use the namespace which we created above just like all other namespaces.

4.6 Constructors and Destructors

4.6.1 Constructors

A constructor is a special member function whose task is to initialize the objects of it's class. A constructor is invoked whenever an object of it's associated class is created. When a class contains a constructor then an object created by that class will be initialized automatically. We pass data to the constructor by enclosing it in the parentheses following the class name when creating an object. In Visual Basic we create constructors by adding a Sub procedure name New to a class. The following code demonstrates the use of constructors in Visual Basic.

```
Imports System.Console
Module Module1

Sub Main()

Dim con As New Constructor(10)
WriteLine(con.display())
'storing a value in the constructor by passing a value(10) and calling it
with the display method
Read()

End Sub

End Module

Public Class Constructor
Public x As Integer

Public Sub New(ByVal value As Integer)
'constructor
x = value
'storing the value of x in constructor
End Sub

Public Function display() As Integer
Return x
'returning the stored value
End Function

End Class
```

4.6.2 Destructors

Coalesce

Destructors run when an object is destroyed. Within a destructor we can place code to clean up the object after it is used. We use *Finalize* method in Visual Basic for this and the Finalize method is called automatically when the .NET runtime determines that the object is no longer required. When working with destructors we need to use the *overrides* keyword with Finalize method as we will override the Finalize method built into the Object class. We normally use Finalize method to deallocate resources and inform other objects that the current object is going to be destroyed. The following code demonstrates the use of Finalize method.

```
Imports System.Console
Module Module1

    Sub Main()
        Dim obj As New Destructor()
    End Sub

End Module

Public Class Destructor

    Protected Overrides Sub Finalize()
        Write("hello")
        Read()
    End Sub

End Class
```

When you run the above code, the word and object, obj of class, destructor is created and "Hello" is displayed. When you close the DOS window, obj is destroyed.

4.7 Structures

Structures can be defined as a tool for handling a group of logically related data items. They are user-defined and provide a method for packing together data of different types. Structures are very similar to Classes. Like Classes, they too can contain members such as fields and methods. The main difference between classes and structures is that classes are *reference* types and structures are *value* types. In practical terms, structures are used for smaller lightweight objects that do not persist for long and classes are used for larger objects that are expected to exist in memory for long periods. We declare a structure in Visual Basic .NET with the **Structure** keyword.

4.7.1 Value Types and Reference Types

Value Types and Reference Types belong to Application data memory and the difference between them is the way variable data is accessed. We will have a brief overview about them.

Value Types

In VB .NET we use the Dim statement to create a variable that represents a value type. For example, we declare a integer variable with the following statement: **Dim x as Integer**. The statement tells the run time to allocate the appropriate amount of memory to hold an integer variable. The statement

Coalesce

creates a variable but does not assign a value to it. We assign a value to that variable like this: **x=55**. When a variable of value type goes out of scope, it is destroyed and its memory is reclaimed.

Reference Types

Creating a variable of reference type is a two-step process, declare and instantiate. The first step is to declare a variable as that type. For example, the following statement **Dim Form1 as new System.Windows.Forms.Form** tells the run time to set enough memory to hold a Form variable. The second step, instantiation, creates the object. It looks like this in code: **Form1=New System.Windows.Forms.Form**. A variable of reference type exists in two memory locations that's why when that variable goes out of scope, the reference to that object is destroyed but the object itself is not destroyed. If any other references to that object exist, the object remains intact. If no references exist to that object then it is subject to garbage collection.

4.7.2 Code for Creating a Structure

The following code creates a Structure named Employee with five fields of different data types.

```
Imports System.Console
Module Module1

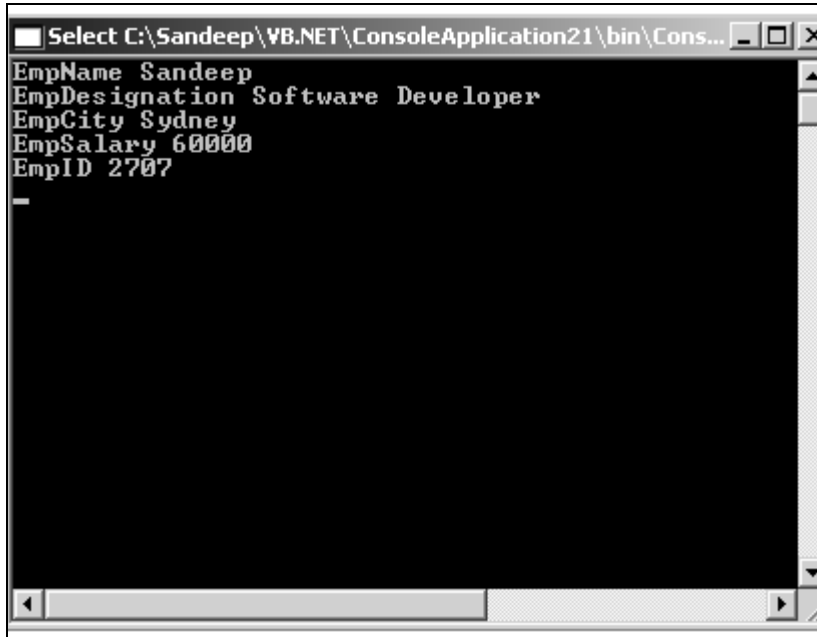
    Structure Employee
        'declaring a structure named Employee
        Dim EmpName As String
        Dim EmpDesignation As String
        Dim EmpCity As String
        Dim EmpSal As Double
        Dim EmpId As Integer
        'declaring five fields of different data types in the structure
    End Structure

    Sub Main()
        Dim san As New Employee()
        'creating an instance of Employee
        san.EmpName = "Sandeep"
        san.EmpDesignation = "Software Developer"
        san.EmpCity = "Sydney"
        san.EmpSal = 60000
        san.EmpId = 2707
        'assigning values to member variables
        WriteLine("EmpName" + " " + san.EmpName)
        WriteLine("EmpDesignation" + " " + san.EmpDesignation)
        WriteLine("EmpCity" + " " + san.EmpCity)
        WriteLine("EmpSalary" + " " + san.EmpSal.ToString)
        WriteLine("EmpID" + " " + san.EmpId.ToString)
        'accessing member variables with the period/dot operator
        Read()
    End Sub

End Module
```

Coalesce

The output of above code is the image below.



4.8 Abstract Classes

An abstract class is the one that is not used to create objects. An abstract class is designed to act as a base class (to be inherited by other classes). Abstract class is a design concept in program development and provides a base upon which other classes are built. Abstract classes are similar to interfaces. After declaring an abstract class, it cannot be *instantiated on its own, it must be inherited*. Like interfaces, abstract classes can specify members that must be implemented in inheriting classes. Unlike interfaces, a class can inherit only one abstract class. Abstract classes can only specify members that should be implemented by all inheriting classes.

4.8.1 Creating Abstract Classes

In Visual Basic .NET we create an abstract class by using the *MustInherit* keyword. An abstract class like all other classes can implement any number of members. Members of an abstract class can either be Overridable (all the inheriting classes can create their own implementation of the members) or they can have a fixed implementation that will be common to all inheriting members. Abstract classes can also specify abstract members. Like abstract classes, abstract members also provide no details regarding their implementation. Only the member type, access level, required parameters and return type are specified. To declare an abstract member we use the *MustOverride* keyword. Abstract members should be declared in abstract classes.

4.8.2 Implementing Abstract Class

When a class inherits from an abstract class, it must implement every abstract member defined by the abstract class. Implementation is possible by overriding the member specified in the abstract class. The following code demonstrates the declaration and implementation of an abstract class.

```
Imports System.Console
Module Module1
```

Coalesce

```
Public MustInherit Class AbstractClass
'declaring an abstract class with MustInherit keyword
Public MustOverride Function Add() As Integer
Public MustOverride Function Mul() As Integer
'declaring two abstract members with MustOverride keyword
End Class

Public Class AbstractOne
    Inherits AbstractClass
'implementing the abstract class by inheriting
Dim i As Integer = 20
Dim j As Integer = 30
'declaring two integers

Public Overrides Function Add() As Integer
Return i + j
End Function
'implementing the add method

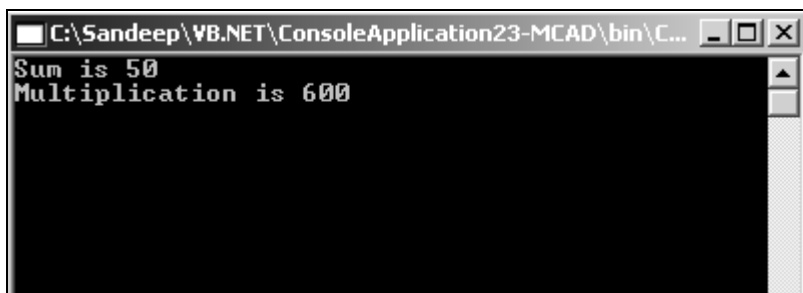
Public Overrides Function Mul() As Integer
Return i * j
End Function
'implementing the mul method

End Class

Sub Main()
Dim abs As New AbstractOne()
'creating an instance of AbstractOne
WriteLine("Sum is" & " " & abs.Add())
WriteLine("Multiplication is" & " " & abs.Mul())
'displaying output
Read()
End Sub

End Module
```

The output of above code is the image below.



Coalesce

Chapter 5

Data Types And Variables

5.1 Data Types in VB.NET

The Data types available in VB.NET and their size are summarized in the table below:

Data Type	Size
Byte	1 byte
Char	2 bytes
Integer	4 bytes
Double	8 bytes
Long	8 bytes
Short	2 bytes
Single	4 bytes
String	Varies
Date	8 bytes
Boolean	2 bytes
Object	4 bytes

Coalesce

Decimal	16 bytes
---------	----------

5.2 Access Specifiers

Access specifiers let's us specify how a variable, method or a class can be used. The following are the most commonly used one's:

Public: Gives variables public access which means that there is no restriction on their accessibility

Private: Gives variable private access which means that they are accessible only within their declaration content

Protected: Protected access gives a variable accessibility within their own class or a class derived from

Friend: Gives variable friend access which means that they are accessible within the program that contains their declaration

Protected Friend: Gives a variable both protected and friend access

Static: Makes a variable static which means that the variable will hold the value even the procedure in which they are declared ends

Shared: Declares a variable that can be shared across many instances and which is not associated with a specific instance of a class or structure

ReadOnly: Makes a variable only to be read and cannot be written

5.3 Variables

Variables are used to store data. A variable has a name to which we refer and the data type, the type of data the variable holds. VB.NET now needs variables to be declared before using them. Variables are declared with the Dim keyword. Dim stands for Dimension.

Example

```
Module Module1
Sub Main()
Dim a,b,c as Integer
'declaring three variables of type integer
a=10
b=20
c=a+b
System.Console.WriteLine("Sum of a and b is" & c)
End Sub
End Module
```

5.4 Enumeration

Coalesce

Enumeration is a related set of constants. They are used when working with many constants of the same type. It's declared with the *Enum* keyword.

Example:

```
Module Module1
Enum Seasons
Summer = 1
Winter = 2
Spring = 3
Autumn = 4
End Enum
Sub Main()
System.Console.WriteLine("Summer is the " & Seasons.Summer &
"season")
End Sub
End Module
```

Output of the above code is Summer is the 1 season. To use a constant from the enumeration it should be referred like this, Seasons.Winter and so on.

5.5 Constants

When we have certain values that we frequently use while programming, we should use Constants. A value declared as constant is of *fixed value* that cannot be changed once set. Constants should be declared as *Public* if we want it to be accessed by all parts of the application. In Visual Basic .NET we use the **Const** keyword to declare a constant. The following line of code declares a constant: **Public Const Pi as Double=3.14159265**

5.6 Operators

Visual Basic comes with many built-in operators that allow us to manipulate data. An operator performs a function on one or more operands. For example, we add two variables with the "+" addition operator and store the result in a third variable with the "=" assignment operator like this: **int x + int y = int z**. The two variables we used in the said line of code (x,y) are called operands. There are various types of operators in Visual Basic and they are described below in the order of their precedence.

5.6.1 Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations that involve calculation of numeric values. The table below summarizes them:

Operators	Description
^	Exponentiation
-	Negation (used to reverse the sign of the given value, exp -intValue)
*	Multiplication
/	Division
\	Integer Division

Coalesce

Mod	Modulus Arithmetic
+	Addition
-	Subtraction

5.6.2 Concatenation Operators

Concatenation operators join multiple strings into a single string. There are two concatenation operators, + and & as summarized below:

Operators	Description
+	String Concatenation
&	String Concatenation

5.6.3 Comparison Operators

A comparison operator compares operands and returns a logical value based on whether the comparison is true or not. The table below summarizes them:

Operators	Description
=	Equality
<>	Inequality
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to

5.6.4 Logical / Bitwise Operators

The logical operators compare Boolean expressions and return a Boolean result. In short, logical operators are expressions which return a true or false result over a conditional expression. The table below summarizes them:

Operators	Description
Not	Negation
And	Conjunction
AndAlso	Conjunction
Or	Disjunction
OrElse	Disjunction
Xor	Disjunction

5.7 Code Samples

5.7.1 Arithmetic Operators

Coalesce

```
Imports System.Console
Module Module1
Sub Main()
Dim x, y, z As Integer
x = 30
y = 20
z = x + y
WriteLine("Addition" & " = " & z)
z = x - y
WriteLine("Subtraction" & " = " & z)
z = x * y
WriteLine("Multiplication" & " = " & z)
z = x / y
WriteLine("Division" & " = " & z)
Read()
End Sub
End Module
```

Output of the above code is the image below.

Coalesce

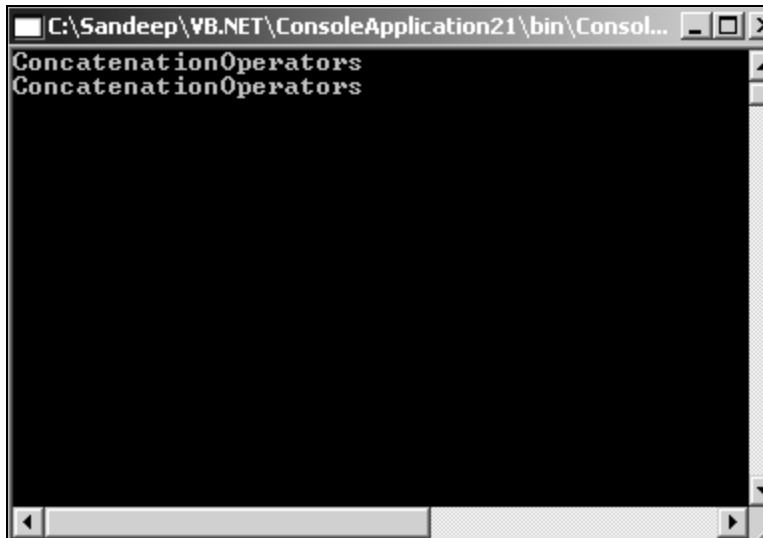


5.7.2 Concatenation Operators

```
Imports System.Console
Module Module1
Sub Main()
Dim str1, str2, str3, str4 As String
str1 = "Concatenation"
str2 = "Operators"
str3 = str1 + str2
WriteLine(str3)
str4 = str1 & str2
WriteLine(str4)
Read()
End Sub
End Module
```

Output of the above code is the image below.

Coalesce



5.7.3 Comparison Operators

```
Imports System.Console
Module Module1
Sub Main()
Dim x, y As Integer
WriteLine("enter values for x and y ")
x = Val(ReadLine())
y = Val(ReadLine())
If x = y Then
WriteLine("x is equal to y")
ElseIf x < y Then
WriteLine("x is less than y")
ElseIf x > y Then
WriteLine("x is greater than y")
End If
Read()
End Sub
End Module
```

Output of the above code is the image below.

Coalesce



5.7.4 Logical / Bitwise Operators

```
Imports System.Console
Module Module1
Sub Main()
Dim x As Boolean
x = Not 45 > 26
WriteLine("x not is false")
' x equals false
x = Not 26 > 87
' x equals True
WriteLine("x not is true")
Read()
End Sub
End Module
```

Output of the above code is the image below.

Coalesce



```
C:\Sandeep\VB.NET\ConsoleApplication21\bin\Consol...
x not is false
x not is true
-
```

Coalesce

Chapter 6

Control Flow Statements

6.1 Conditional Statements

Working with the *If...Else* statement

If conditional expression is one of the most useful control structures which allows us to execute a expression if a condition is true and execute a different expression if it is False. The syntax looks like this:

```
If condition Then
[statements]
Else If condition Then
[statements]
-
-
Else
[statements]
End If
```

Understanding the Syntax:

If the condition is true the statements following the Then keyword will be executed, else the statements following the ElseIf will be checked and if true, will be executed, else the statements in the else part will be executed.

Working with an example

```
Module Module1
Sub Main()
Dim i As Integer
System.Console.Write("enter an integer, 1 or 2 or 3")
i = Val(System.Console.ReadLine())
'ReadLine() method is used to read from console
If i = 1 Then
System.Console.Write("Pass")
ElseIf i = 2 Then
System.Console.Write("Fail")
Else
System.Console.Write("No Result")
End If
End Sub
End Module
```

In the above example the console reads the value typed and checks it against the condition and displays the statements when the condition is matched.

Coalesce

6.2 The *For* Loop

The For loop is the most popular loop. For loops enables us to execute a series of expressions multiple numbers of times . The For loop in VB.NET needs a loop index, which counts the number of loop iterations as the loop executes. The syntax for the For loop looks like this:

```
For index=start to end[Step step]
[statements]
[Exit For]
[statements]
Next[index]
```

The index variable is set to start automatically when the loop starts. Each time in the loop, index is incremented by step and when index equals end, the loop ends.

Example on For loop

```
Module Module1
Sub Main()
Dim d As Integer
For d = 0 To 2
System.Console.WriteLine("In the For Loop")
Next d
End Sub
End Module
```

The output of above code is, "In the For Loop" being displayed three times.

6.3 The *While* loop

While loop keeps executing until the condition against which it tests remain true. The syntax of while loop looks like this:

```
While condition
[statements]
End While
```

Example on While loop

```
Module Module1
Sub Main()
Dim d, e As Integer
d = 0
e = 6
While e > 4
e -= 1
d += 1
```

Coalesce

```
End While
System.Console.WriteLine("The Loop ran " & e & "times")
End Sub
End Module
```

6.4 The *Do* Loop

The Do loop can be used to execute a fixed block of statements indefinite number of times. The Do loop keeps executing it's statements while or until the condition is true. Two keywords *while* and *until* can be used with the do loop. The Do loop also supports an Exit Do statement which makes the loop to exit at any moment. The syntax of Do loop looks like this:

```
Do[{while | Until} condition]
[statements]
[Exit Do]
[statements]
Loop
```

Example on Do loop

```
Module Module1
Sub Main()
Dim str As String
Do Until str = "Cool"
System.Console.WriteLine("What to do?")
str = System.Console.ReadLine()
Loop
End Sub
End Module
```

The output of the above program when executed keeps on asking "What to do?" until the word "Cool" is typed.

Coalesce

Chapter 7

Arrays and String manipulation

7.1 Arrays

Arrays are programming constructs that store data and allow us to access them by *numeric index or subscript*. Arrays helps us create shorter and simpler code in many situations. Arrays in Visual Basic .NET inherit from the *Array* class in the System namespace. They are declared using **Dim**, **ReDim**, **Static**, **Private**, **Public** and **Protected** keywords. An array can have one dimension or more than one. The dimensionality of an array refers to the number of subscripts used to identify an individual element. In Visual Basic we can specify up to 32 dimensions. Arrays do not have fixed size in Visual Basic. We can change the size of an array after you have created it. The ReDim statement assigns a completely new array object to the specified array variable. Therefore, we can use ReDim to change the length of each dimension.

We declare a one-dimensional array as: **Dim Test(10)**. This Test array has 10 elements, starting from Test(0) to Test(9). The first element of an array is always referred by zero index. We declare a two-dimensional array as: **Dim Test1(10, 20)**. To change the dimension (redimension) of the Test array which we already declared we use the statement: **ReDim Test(20)**. The size of Test array changes from 10 to 20.

The following code demonstrates the usage of arrays.

```
Imports System.Console

Module Module1
Sub Main()
Dim sport(5) As String
'declaring an array
sport(0) = "Soccer"
sport(1) = "Cricket"
sport(2) = "Rugby"
sport(3) = "Aussie Rules"
sport(4) = "BasketBall"
'storing values in the array
WriteLine("Name of the Sport in the third location" & " " &
sport(2))
'displaying value from array
Read()
End Sub
End Module
```

The above code declared a Sport array to hold five values. Five values are stored in the array starting from index (0) to index (4). When you run the above code, the output displayed is the image below.

Coalesce



7.2 Strings

Strings in Visual Basic are supported by the .NET *String* class. The *String* data type can represent a series of characters and can contain approximately up to 2 billion *Unicode* characters. There are many built-in functions in the *String* class. Some .NET Framework functions are also built into the *String* class. The following code puts some *String* functions to work.

```
Imports System.Console

Module Module1
Sub Main()
Dim string1 As String = "Strings in Visual Basic"
Dim string2 As String
Dim string3 As String
Dim string4 As String
Dim string5 As String
Dim string6 As String
string2 = UCase(string1)
'converts string to uppercase
string3 = LCase(string1)
'converts string to lowercase
string4 = string1.Substring(11, 6)
'returns a substring
string5 = string1.Clone
'clones a string
string6 = string1.GetHashCode
'gets the hashcode
WriteLine(string2)
WriteLine(string3)
WriteLine(string4)
```

Coalesce

```
WriteLine(string5)
WriteLine(string6)
Read()
End Sub
End Module
```

Output of above code is the image below.



7.3 Math Functions

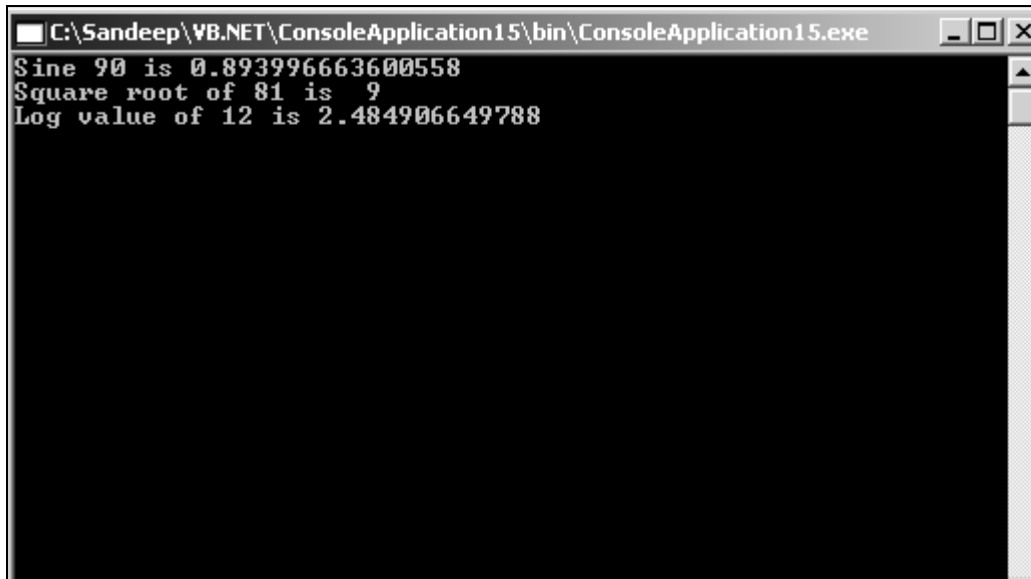
Visual Basic also provides support for handling *Mathematical calculations*. Math functions are stored in *System.Math* namespace. We need to import this namespace when we work with Math functions. The functions built-into Math class helps us calculate the Trigonometry values, Square roots, logarithm values, etc. The following code puts some Math functions to work

```
Imports System.Console
Imports System.Math

Module Module1
Sub Main()
WriteLine("Sine 90 is" & " " & Sin(90))
'display Sine90 value
WriteLine("Square root of 81 is" & " " & Sqrt(81))
'displays square root of 81
WriteLine("Log value of 12 is" & " " & Log(12))
'displays the logarithm value of 12
Read()
End Sub
End Module
```

Coalesce

Output of above code is the image below.



7.4 Converting between Data types

We can convert a data type from one type to other. Let's do that in code. The example below declares two variable, one of type double and the other integer. The double data type is assigned a value and is converted to integer type. When you run the code, the result displayed is an integer value, in this case the value displayed is 132 instead of 132.31223.

```
Module Module1
Sub Main()
Dim d=132.31223 as Double
Dim i as Integer
i=d
System.Console.WriteLine("Integer value is" & i)
End Sub
End Module
```

Using *CType function* for conversion

If we are not sure of the name of a particular conversion function, we can use the CType function. The example above with a CType function looks like this:

```
Module Module1
Sub Main()
Dim d As Double
d = 132.31223
Dim i As Integer
i = CType(d, i)
'two arguments, type we are converting from, to type desired
```


Coalesce

```
System.Console.WriteLine("Integer value is" & i)
End Sub
End Module
```

Below is the list of conversion functions which we can use in VB .NET.

CBool - use this function to convert to Bool data type
CByte - use this function to convert to Byte data type
CChar - use this function to convert to Char data type
CDate - use this function to convert to Date type
CDBl - use this function to convert to Double data type
CDec - use this function to convert to Decimal data type
CInt - use this function to convert to Integer data type
CLng - use this function to convert to Long data type
CObj - use this function to convert to Object type
CShort - use this function to convert to Short data type
CSng - use this function to convert to Single data type
CString - use this function to convert to String data type

Attributes

Attributes are those, that lets us specify information about the items we are using in VB.NET. Attributes are enclosed in angle brackets(< >) and are used when VB.NET needs to know more beyond the standard syntax.

7.5 Date and Time Functions

Visual Basic also provides support for handling *Date and Time values*. Date and Time functions are stored in *System.DateTime* namespace. We need to import this namespace when we work with DateTime functions. The functions built-into DateTime class helps us calculate the date difference, year, month, day and etc. The following code puts some DateTime functions to work

```
Imports System.Console
Imports System.DateTime

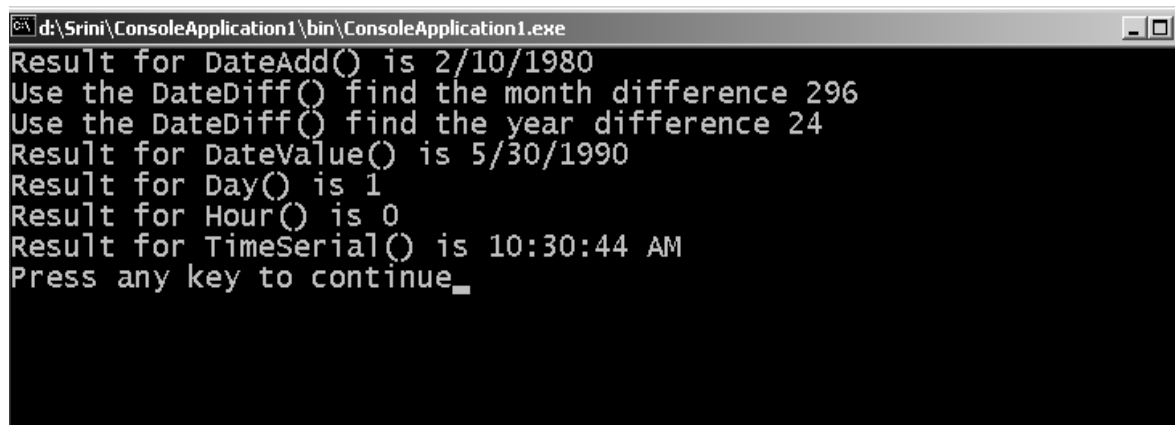
Module Module1

    Sub Main()
        Dim dt1 As Date
        Dim dt2 As Date
        Dim Res As Date
        Dim Ret As Long
        dt1 = "01/01/1980"
        Res = DateAdd(DateInterval.Day, 40, dt1)
        WriteLine("Result for DateAdd() is " & Res)
        dt2 = Now()
        Ret = DateDiff(DateInterval.Month, dt1, dt2)
        WriteLine("Use the DateDiff() find the month
```

Coalesce

```
difference " & Ret)
Ret = DateDiff(DateInterval.Year, dt1, dt2)
WriteLine("Use the DateDiff() find the year
difference " & Ret)
Res = DateValue("05/30/1990")
WriteLine("Result for DateValue() is " & Res)
Ret = dt1.Day
WriteLine("Result for Day() is " & Ret)
Ret = dt1.Hour
WriteLine("Result for Hour() is " & Ret)
Res = TimeSerial(10, 30, 44)
WriteLine("Result for TimeSerial() is " & Res)
End Sub
End Module
```

Output of above code is the image below.



The screenshot shows a console window titled "d:\Srin\\ConsoleApplication1\bin\ConsoleApplication1.exe". The output text is as follows:

```
Result for DateAdd() is 2/10/1980
Use the DateDiff() find the month difference 296
Use the DateDiff() find the year difference 24
Result for DateValue() is 5/30/1990
Result for Day() is 1
Result for Hour() is 0
Result for TimeSerial() is 10:30:44 AM
Press any key to continue_
```

Coalesce

Chapter 8

Procedures

8.1 Subroutines

Procedures are a series of statements that are executed when called. Procedures makes us handle the code in a simple and organized fashion. Sub procedures are procedures which do not return a value. Each time when the Sub procedure is called, the statements within it are executed until the matching End Sub is encountered. The Sub Main(), which is the starting point of the program itself is a sub procedure. When the application starts execution, the control is transferred to Main Sub procedure automatically which is called by default.

Example of a Sub Procedure

```
Module Module1
Sub Main()
'sub procedure Main() which is called by default
Display()
'sub procedure display() which we are creating
End Sub
Sub Display()
System.Console.WriteLine("Using Sub Procedures")
'executing sub procedure Display()
End Sub
End Module
```

Looking at the example above, we have a sub procedure called Display() within the main method which is called when the control passes into the main method. The output displayed is "Using Sub Procedures".

8.2 Functions

Function is a Procedure which returns a value. Functions are used to evaluate data, make calculations with it, or to transform data. Declaring a Function is similar to declaring a Sub procedure but are declared with the *Function* keyword as with the *Sub* keyword for the Sub procedure. The following is an example on Functions:

```
Private Sub Form8_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
MessageBox.Show("Sum is " & Add())
'using the MessageBox to show the sum of two numbers
End Sub
Function Add() As Integer
'declaring a function add
Dim i, j As Integer
'declaring two integers and assigning values to them
```

Coalesce

```
i = 10  
j = 20  
Return (i + j)  
'performing the sum of two integers and returning it's value  
End Function  
End Class
```

8.3 Commenting the Code

Comments in VB.NET begin with a single quote(') character and the statements following that are ignored by the compiler. Comments are generally used to specify what is going on in the program and also gives an idea about the flow of the program.

Coalesce

Chapter 9

Exception Handling

Exceptions are runtime errors that occur when a program is running, causing the program to abort without execution. Such kind of situations can be handled using Exception Handling. By placing specific lines of code in the application, we can handle most of the errors that we may encounter and enable the application to continue running. VB.NET supports two types of exception handling namely *Unstructured* exception Handling using the on error goto statement and *Structured* exception handling using Try....Catch.....Finally

Let's look at the new kind of exception handling introduced in VB.NET which is the Structured Exception Handling.

VB.NET uses Try....Catch....Finally block type exception handling. The syntax looks like this:

```
Module Module1
Sub Main()
Try
-
-
Catch e as Exception
-
-
Finally
End Try
End Sub
End Module
```

Example

```
Module Module1
Sub Main()
Dim a = 0, b = 1, c As Integer
Try
c = b / a
'the above line throws an exception
System.Console.WriteLine("C is " & c)
Catch e As Exception
System.Console.WriteLine(e)
'catching the exception
End Try
System.Console.Read()
End Sub
End Module
```

Coalesce

The output of the above code displays a message stating the exception. The reason for the exception is because any number divided by zero is infinity. When working with Structured exception handling you can have multiple Catch blocks to handle different types of exceptions differently. The code in the Finally block is optional. If there is a Finally block in the code then that code is executed last.

Coalesce

Chapter 10

Statements and Scope

10.1 Statements

A statement is a complete instruction. It can contain keywords, operators, variables, literals, expressions and constants. Each statement in Visual Basic should be either a declaration statement or a executable statement. A *declaration statement* is a statement that can create a variable, constant, data type. They are the one's we generally use to declare our variables. On the other hand, *executable statements* are the statements that perform an action. They execute a series of statements. They can execute a function, method, loop etc.

10.1.1 Option Statement

The Option statement is used to set a number of options for the code to prevent syntax and logical errors. This statement is normally the first line of the code. The Option values in Visual Basic are as follows.

Option Compare: You can set it's value to *Text* or *Binary*. This specifies if the strings are compared using binary or text comparison operators.

Option Explicit: Default is *On*. You can set it to *Off* as well. This requires to declare all the variables before they are used.

Option Strict: Default is *Off*. You can set it to *On* as well. Used normally when working with conversions in code. If you want to assign a value of one type to another then you should set it to *On* and use the conversion functions else Visual Basic will consider that as an error.

The following code does not perform anything special but will show where to place the Option statement.

```
Option Strict Off
Imports System
Module Module 1
Sub Main ()
Console.WriteLine ("Using Option")
End Sub
End Module
```

We always should program with Option Strict On. Doing so allows us to catch many errors at compile time that would otherwise be difficult to track at run time.

Coalesce

10.2 Imports Statement

The Imports statement is used to import namespaces. Using this statement prevents you to list the entire namespace when you refer to them.

Example of Imports Statement

The following code imports the namespace *System.Console* and uses the methods of that namespace thus preventing us to refer to it every time we need a method of this namespace.

```
Imports System.Console
Module Module1
Sub Main()
Write("Imports Statement")
WriteLine("Using Import")
End Sub
End Module
```

The two methods used above without an imports statement would look like this: `System.Console.Write("Imports Statement")` and `System.Console.WriteLine("Using Import")`

10.3 Boxing

Boxing is the implicit conversion of value types to reference types. Recall that all classes and types are derived from the *Object* class. Because they are derived from *Object* class, they can be implicitly converted to that type. The following sample shows that:

```
Dim x as Integer=20
'declaring an integer x
Dim o as Object
'declaring an object
o=x
converting integer to object
```

Unboxing, on the other hand is the conversion of a boxed value back to a value type.

Scope

The scope of an element in code is all the code that can refer to it without qualifying its name. Stated other way, an element's scope is its accessibility in code. Scope is normally used when writing large programs as large programs divide the code into different classes, modules etc. Also, scope prevents the chance of code referring to the wrong item. The different kinds of scope available in VB.NET are as follows:

Block Scope: The element declared is available only within the code block in which it is declared.

Procedure Scope: The element declared is available only within the procedure in which it is declared.

Module Scope: The element is available to all code within the module and class in which it is declared.

Namespace Scope: The element declared is available to all code in the namespace.

Coalesce

PART II

Windows Applications

Coalesce

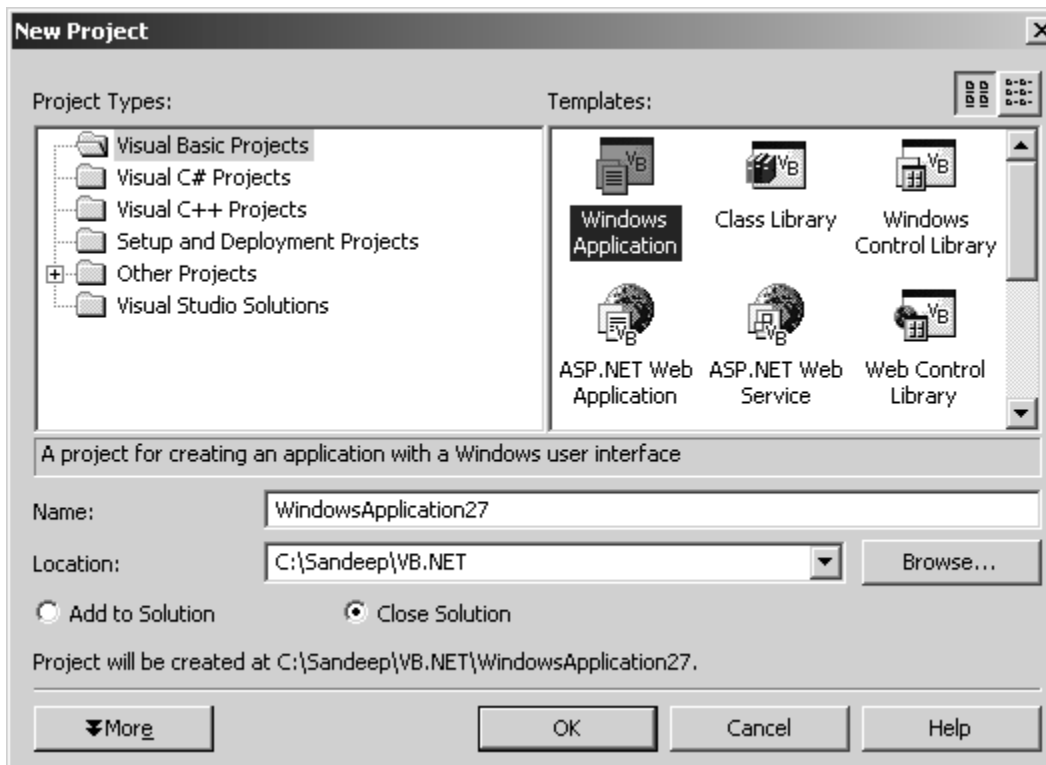
Chapter 11

Win Forms

11.1 Using Forms

In Visual Basic, its these Forms with which we work. They are the base on which we build, develop all our user interface and which comes with a rich set of classes. Forms allow us to work visually with controls and other items from the toolbox. In VB.NET, forms are based on the *System.Windows.Forms* namespace and the form class is *System.Windows.Forms.Form*. The form class is based on the *Control* class which allows it to share many properties and methods with other controls.

When we open a new project in Visual Basic the box that appears first is the one which looks like the image below. Since we are working with Windows Applications (Forms) you need to select WindowsApplication and click OK.

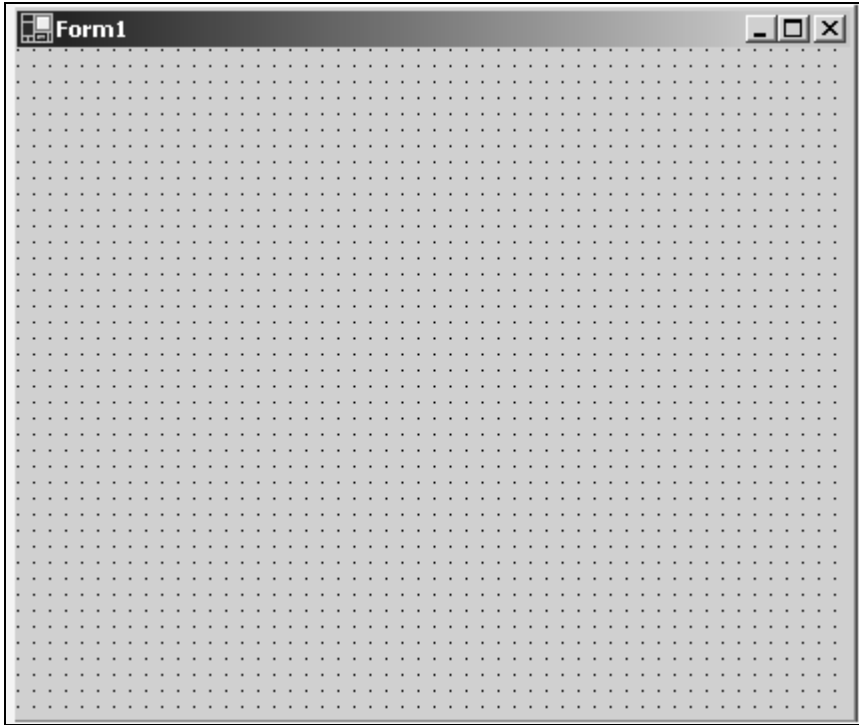


Once you click OK, a new Form opens with the title, Form1, towards the top left side of the form and maximize, minimize and close buttons towards the top right of the form. The whole form is surrounded with a border. The main area of the form in which we work is called the *Client Area*. It's in this client area we design the user interface by working with controls leaving all the code to the code behind file. Forms also support events which let's the form know that something happened

Coalesce

with the form, for example when we double-click on the form, the Form load event occurs. VB.NET also supports forms to be inherited.

Image of a Windows Form



Typically the Form looks like this in Code which is handled by the Framework.

```
Public Class Form1
Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

Public Sub New()
MyBase.New()

"This call is required by the Windows Form Designer.
InitializeComponent()

'Add any initialization after the InitializeComponent() call

End Sub

'Form overrides dispose to clean up the component list.
Protected Overloads Overrides Sub Dispose(ByVal disposing As
Boolean)
If disposing Then
If Not (components Is Nothing) Then
components.Dispose()
End If
```

Coalesce

```
End If
MyBase.Dispose(disposing)
End Sub

'Required by the Windows Form Designer
Private components As System.ComponentModel.IContainer

'NOTE: The following procedure is required by the Windows Form
Designer
'It can be modified using the Windows Form Designer.
'Do not modify it using the code editor.
<System.Diagnostics.DebuggerStepThrough> Private Sub
InitializeComponent()
'
'Form1
'
Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
Me.ClientSize = New System.Drawing.Size(496, 493)
Me.Name = "Form1"
Me.Text = "Form1"

End Sub

#End Region

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

End Sub
End Class
```

11.2 Working With Forms

Well, let's now start working with Forms. Once when you open a new form you can have a look at the default properties of the form by selecting *View->Properties Window* or by pressing *F4* on the keyboard. The properties window opens with the default properties set to the form by the software.

Briefly on Properties:

Appearance

Appearance section of the properties window allows us to make changes to the appearance of the form. With the help of appearance properties we can have a background color, background image for the entire form, set the border style for the form from a predefined list, change the cursor, set the font for the text on the form and so on.

Behavior

Coalesce

Notable Behavior property is the *enabled* property which lets us enable/disable the form by setting the property to True/False.

Layout

Layout properties are all about the structure of the form. With these properties we can set the location of the form, maximum size of the form, minimum size of the form, exact size of the form with the size property when designing. Note the property start position, this property lets you specify the location of the form where it should appear when you run the application which you can select from a predefined list.

Window Style

The notable property under this is the *ControlBox* property which by default it is set to True. Setting the property to False makes the minimize, maximize and cancel buttons on the top right side of the form invisible.

An Example:

You can run the Form by selecting *Debug->Start* from the main menu or by pressing *F5* on the keyboard. When you run a blank form with no controls on it, nothing is displayed. It looks like the image below.



Now, add a TextBox and a Button to the form from the toolbox. The toolbox can be selected from *View->ToolBox* under the main menu or by holding *Control+Alt+X* on the keyboard. Once adding the TextBox and Button is done, run the form. The output window displays a TextBox and a Button. When you click the Button, nothing happens. We need to write an event for the Button stating

Coalesce

something should happen when you click it. To do that, get back to the design of the form and double-click on the Button. Doing so opens an event handler for the Button where you specify what should happen when you click the button. That looks like this in code.

```
Public Class Form1
Inherits System.Windows.Forms.Form
#Region " Windows Form Designer generated code "
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
End Sub
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Button1.Click
End Sub
End Class
```

Place the code `TextBox1.Text="Welcome to Forms"`, in the Click event of the Button. It looks like this in code.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Button1.Click
TextBox1.Text="Welcome to Forms"
End Sub
```

Run the application now and when you click the Button, the output "Welcome to Forms" is displayed in the TextBox.

Alternatively, you can use the *MsgBox* or *MessageBox* functions to display text when you click on the Button. To do that place a Button on the form and double-click on that which opens the event handler for the button. In that write this line of code, `MsgBox("Welcome to Forms")` or `MessageBox.Show("Welcome to Forms")`

It looks like this in code.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Button1.Click
MsgBox("Welcome to Forms")
End Sub

or

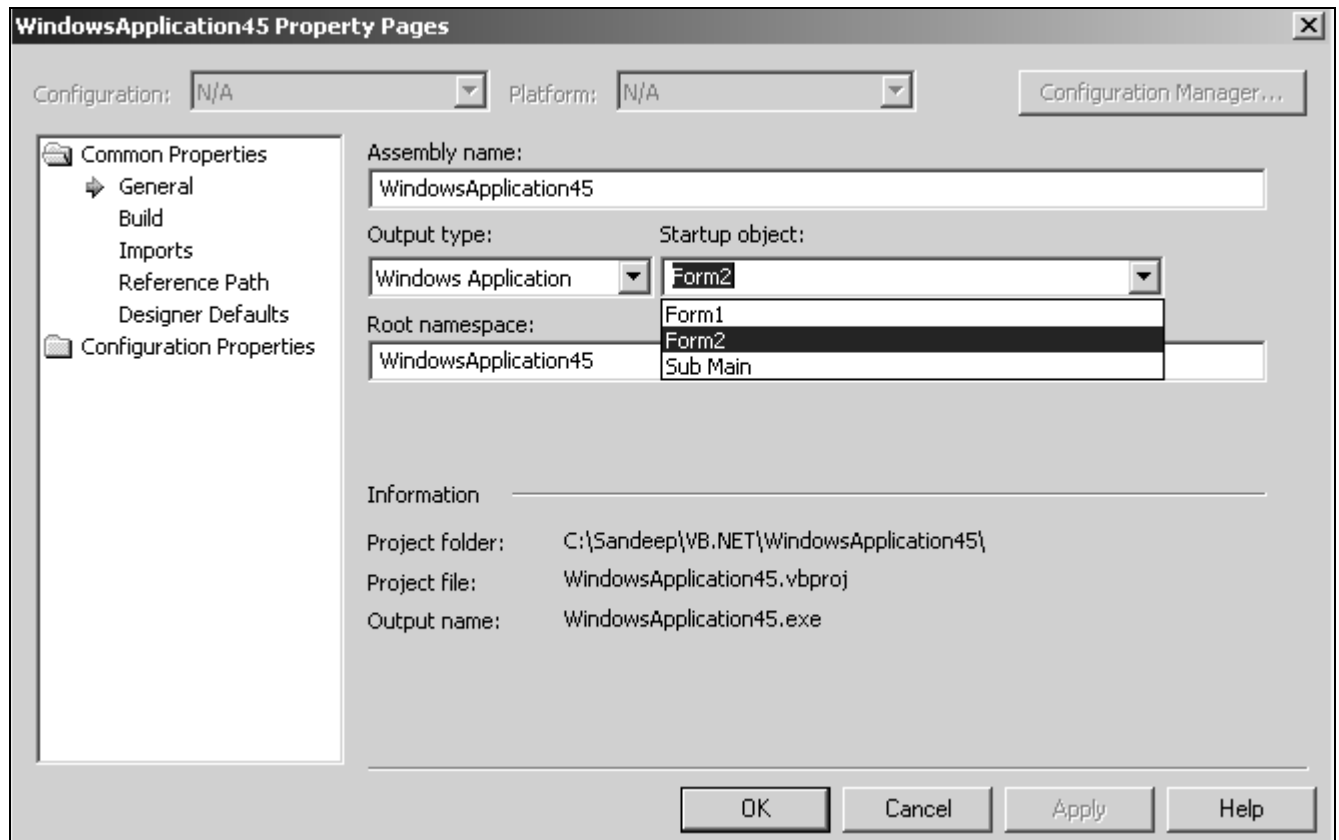
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Button1.Click
MessageBox.Show("Welcome to Forms")
End Sub
```

When you run the form and click the Button a small message box displays, "Welcome to Forms".

Coalesce

11.3 Adding a New Form to the Project

You can add a new form to the project with which you are working. To do that select *File->Add New Item* from the main menu which displays a window asking you what to add to the project. Select *Windows Form* from the window and click Open to add a new form to the project. Alternatively, you can add a new form with the Solution Explorer. To add a new form with the Solution Explorer, right-click on the project name in Solution Explorer and Select *Add-> Add Windows Form*. Once you finish adding a new form, if you want the form which you added to be displayed when you you run the application you need to make some changes. You need to set the new form as *Startup Object*. To do that right-click on the project name in Solution Explorer window and select *Properties* which displays the *Property Pages window*. On this window click the drop down box which is labeled as *Startup Object*. Doing that displays all the forms available in the project. It looks like the image below.

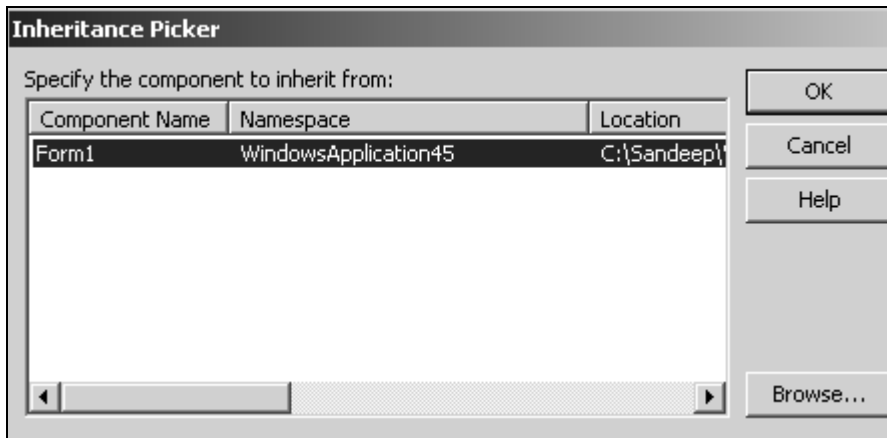


Select the form which you want to be displayed when you run the program and click Apply. Now, when you run the application, the form with which you want to work will be displayed.

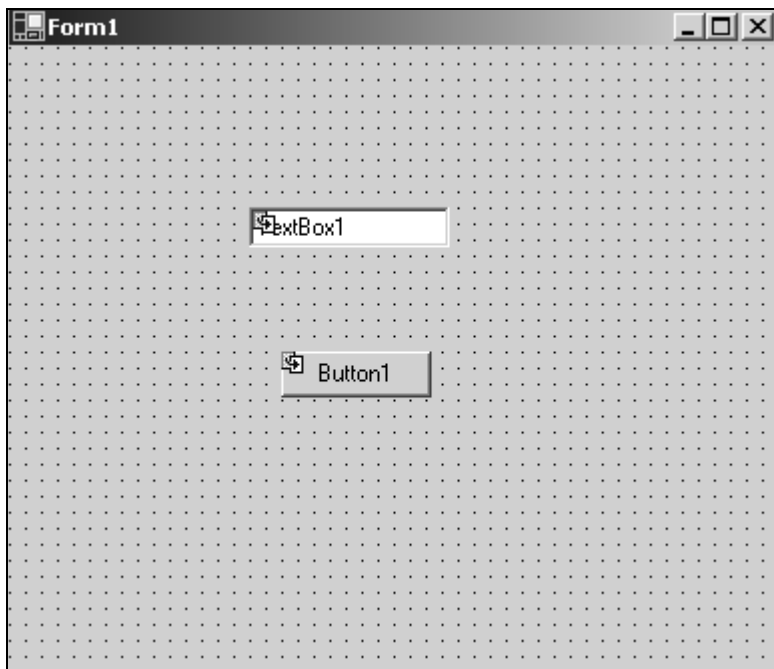
11.4 Visual Inheritance with Forms

As we know, Inheritance allows us to derive one class from another, with VB.NET we can inherit one form from another. Let's see how we can inherit a new form from an existing form. Say, we have a form named Form1 with some controls on it and we want to inherit a new form from that form. To derive a new form from the existing one select *Project->Add Inherited Form* from the main menu which opens up the Add New Item window. Double-click on the *Inherited Form Icon* in the templates box to open the Inheritance Picker. Inheritance Picker window looks like the image below.

Coalesce



Select Form1 in the picker and click OK. Clicking OK adds a new form, Form2 which is derived from Form1. Everything on Form2 including the title is copied from Form1 but the only difference, the controls on Form2 come with a special icon at the upper left corner which indicates that the form is inherited and the controls are locked. The image below displays a Inherited Form.



11.5 Working with Multiple Forms

In this section we will work with multiple forms. Say, for example, we want to open a new form (Form2) when a button in the current form (Form1) is clicked. To do that, open a new project by selecting *File->New->Project->Visual Basic->Windows Application*. This adds a form (Form1) to the application. Now, add a new form (Form2) by selecting *File->Add New Item->Windows Form* and name it as Form2. Now, drag a button (Button1) onto Form1. We want to open Form2 when the button in Form1 is clicked. Unlike earlier versions of Visual Basic, VB.NET requires us to refer to Form2 in Form1 in order to open it i.e creating an object of Form2 in Form1. The code for that looks like this:

Coalesce

```
Public Class Form1 Inherits System.Windows.Forms.Form
Dim other As New Form2()
'Creating a reference to Form2
'Windows Form Designer Generated Code
Private Sub Button1_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) _
    Handles Button1.Click
other.Show()
End Sub
```

11.6 Handling Mouse Events in Forms

We can also handle mouse events such as mouse pointer movements, etc in Forms/Controls. The mouse events supported by VB.NET are as follows:

MouseDown: This event happens when the mouse pointer is over the form/control and is pressed

MouseEnter: This event happens when the mouse pointer enters the form/control

MouseUp: This event happens when the mouse pointer is over the form/control and the mouse button is released

MouseLeave: This event happens when the mouse pointer leaves the form/control

MouseMove: This event happens when the mouse pointer is moved over the form/control

MouseWheel: This event happens when the mouse wheel moves while the form/control has focus

MouseHover: This event happens when the mouse pointer hovers over the form/control

The properties of the MouseEventArgs objects that can be passed to the mouse event handler are as follows:

Button: Specifies that the mouse button was pressed

Clicks: Specifies number of times the mouse button is pressed and released

X: The X-coordinate of the mouse click

Y: The Y-coordinate of the mouse click

Delta: Specifies a count of the number of detents (rotation of mouse wheel) the mouse wheel has rotated

11.7 Working with an Example

We will work with an example to check the mouse events in a form. We will be working with MouseDown, MouseEnter and MouseLeave events in this example. Now, Drag three TextBoxes (TextBox1, TextBox2, TextBox3) onto the form. The idea here is to display the results of these events in the TextBoxes.

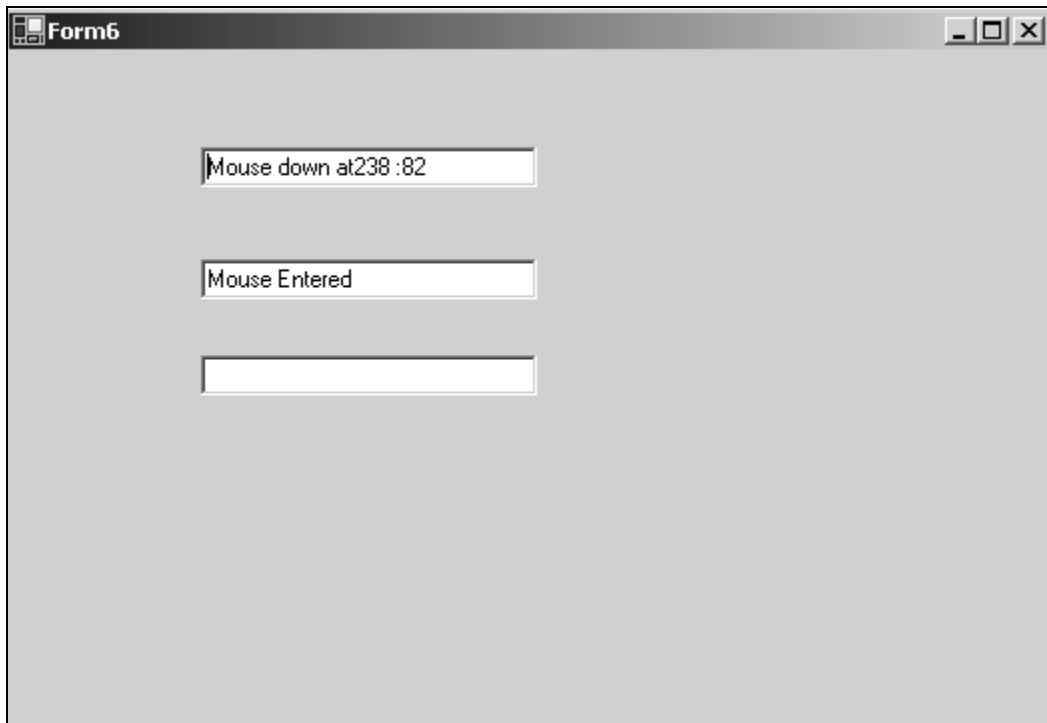
The code for that looks like this:

```
Public Class Form1 Inherits System.Windows.Forms.Form
'Windows Form Designer Generated Code
Private Sub Form1_Mousedown(ByVal sender As System.Object,
ByVal e As _
System.Windows.Forms.MouseEventArgs) Handles
MyBase.MouseDown
If e.Button = MouseButton.Left Then
TextBox1.Text = "Mouse down at" + CStr(e.X) + " :" + CStr(e.Y)
```

Coalesce

```
'displaying the coordinates in TextBox1 when a mouse is pressed on  
the form  
End If  
End Sub  
  
Private Sub Form1_MouseEnter(ByVal sender As System.Object,  
ByVal e As System.EventArgs) Handles MyBase.MouseEnter  
    TextBox2.Text = "Mouse Entered"  
    'display mouse entered in the TextBox2 when the mouse pointer  
    enters the form  
End Sub  
  
Private Sub Form1_MouseLeave(ByVal sender As System.Object,  
ByVal e As System.EventArgs) Handles MyBase.MouseLeave  
    TextBox3.Text = "Mouse Exited"  
    'display mouse exited in the Textbox3 when the mouse pointer leaves  
    the form  
End Sub  
  
End Class
```

The output of above code looks like the image below.



11.8 Beep Function

The Beep Function in VB.NET can be used to make the computer emit a beep. To see how this works, drag a Button onto the form and place the following code in the button click event:

Coalesce

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e  
As System.EventArgs)_  
Handles Button1.Click  
Beep()  
End Sub
```

Now, when you click the button, the computer emits a beep.

Coalesce

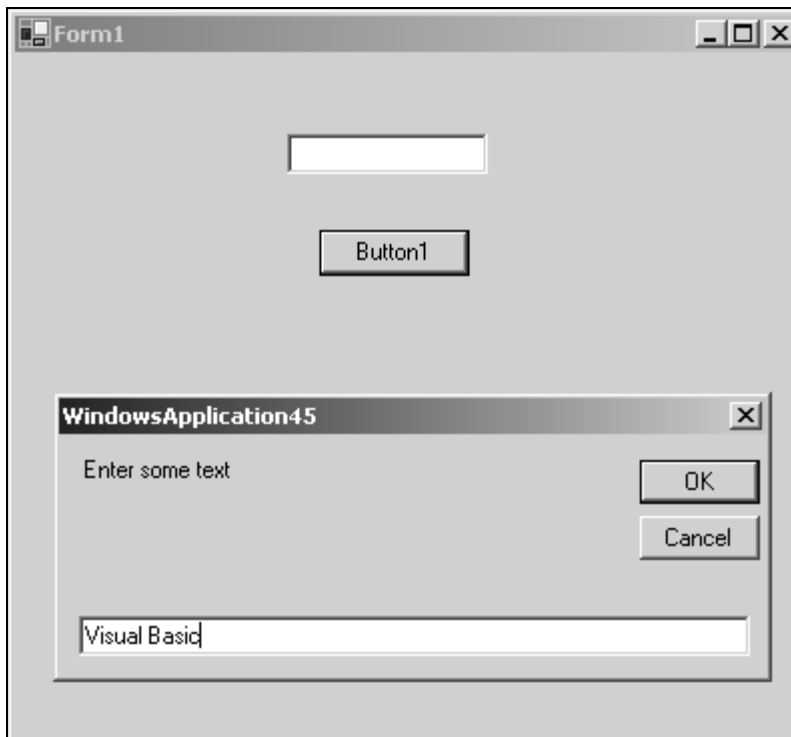
Chapter 12

More On Windows forms

12.1 InputBox

InputBox function gets a string of text entered by the user. They are similar to the JavaScript prompt windows that ask us to enter some text and performs some action after the OK button is clicked. In Visual Basic, Input boxes let you display a prompt, read the text entered by the user and returns a string result. The following code demonstrates InputBox functionality. Drag a Button and TextBox control from the toolbox onto the form. When you click the Button, InputBox asks to enter some text and after you click OK it displays the text you entered in the TextBox. The image below displays output.

```
Private Sub Button1_Click(ByVal sender As System.Object,  
ByVal e As System.EventArgs)_  
Handles Button1.Click  
Dim s As String = InputBox("Enter some text")  
'storing the text entered in a string  
TextBox1.Text = s  
'displaying string in the textbox  
End Sub
```



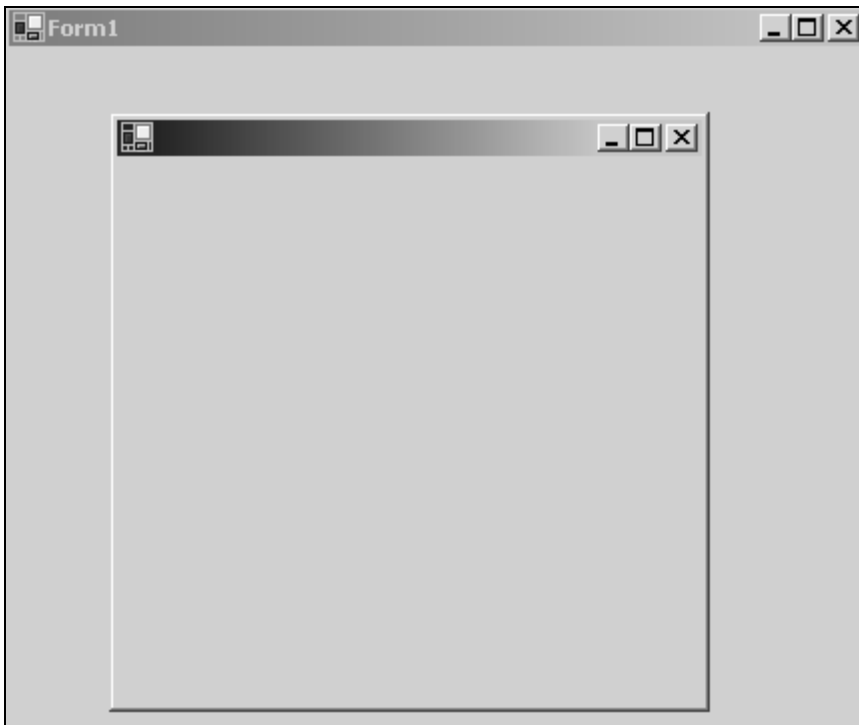
Coalesce

12.2 Owned Forms

Visual Basic also allows us to create owned forms. An owned form is tied to the form that owns it. If you minimize the owner form, the owned form will also be minimized, if you close the owner form, the owned form will be closed and so on. Owned Forms are added to a form with the *AddOwnedForm* method and removed with the *RemoveOwnedForm* method. The following code demonstrates the creation of owned forms. Drag a Button from the toolbox onto the form (form1). When you click the button, Form1 will make an object of the Form class into an owned form and displays (image below) it. The code for that looks like this:

```
Public Class Form1 Inherits System.Windows.Forms.Form
Dim OwnedForm1 As New Form()

Private Sub Button2_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs)_
Handles Button2.Click
Me.AddOwnedForm(OwnedForm1)
'adding an owned form to the current form
OwnedForm1.show()
'displaying the owned form
End Sub
```



Coalesce

12.3 Anchoring and Docking

Docking and Anchoring is used to make controls cover the whole client area of a form. When you dock a window, it adheres to the edges of its container (form). To dock a particular control, select the *Dock* property of that control from the properties window. Selecting the dock property opens up a small editor from which you can select to which side on the form should the control be docked.

12.3.1 Docking a control

Select the control that you want to dock. In the Properties window select *Dock* property. An editor is displayed that shows a series of boxes representing the edges and the center of the form. Click the button that represents the edge of the form where you want to dock the control. To fill the contents of the control's form or container control, click the center box. The control is automatically resized to fit the boundaries of the docked edge.

12.3.2 Anchoring

Anchoring is used to resize controls dynamically with the form. If you are designing a form that the user can resize at run time, the controls on your form should resize and reposition properly. The *Anchor* property defines an anchor position for the control. When a control is anchored to a form and the form is resized, the control maintains the distance between the control and the anchor positions.

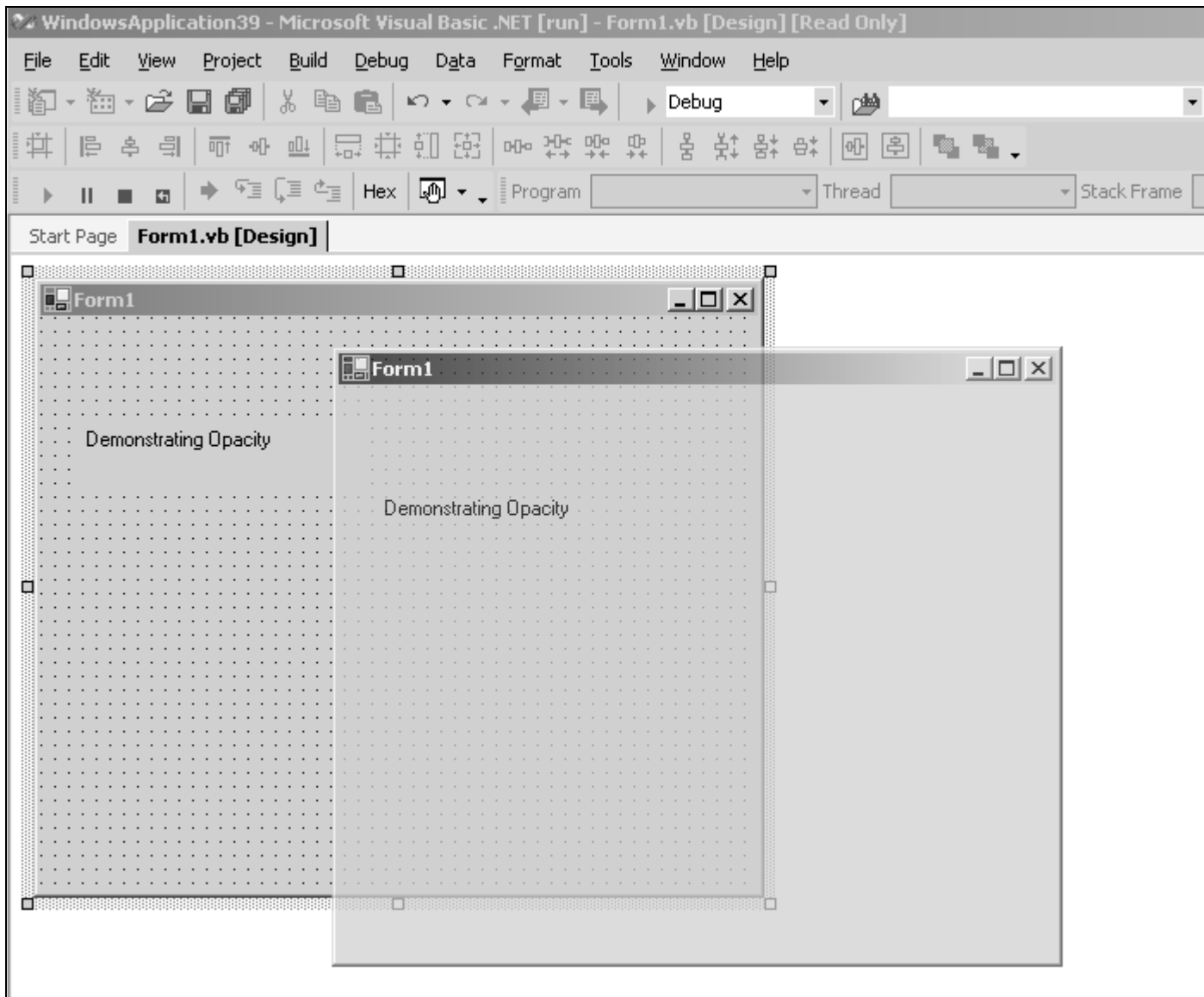
Anchoring a control

To anchor a control, select the control that you want to anchor. In the Properties window, select *Anchor* property. Selecting that displays an editor that shows a cross. To set an anchor, click the top, left, right, or bottom section of the cross. By default controls are anchored to the top and left side. To clear a side of the control that has been anchored, click that side of the cross. When your run the form , the control resizes to remain positioned at the same distance from the edge of the form. The distance from the anchored edge always remains the same as the distance defined when the control is positioned during design time.

12.3.3 Windows Forms Opacity

The *Opacity* property is used to make a form transparent. To set this property set the *Opacity* property to a value between 0 (complete transparency) and 100 (complete opacity) in the properties window. The Image below demonstrates Opacity property. The form is set to an opacity value of 75.

Coalesce



Transparent forms are only supported in Windows 2000 or later. Windows Forms will be completely opaque when run on older operating systems, such as Windows 98, regardless of the value set for the Opacity property.

Coalesce

Chapter 13

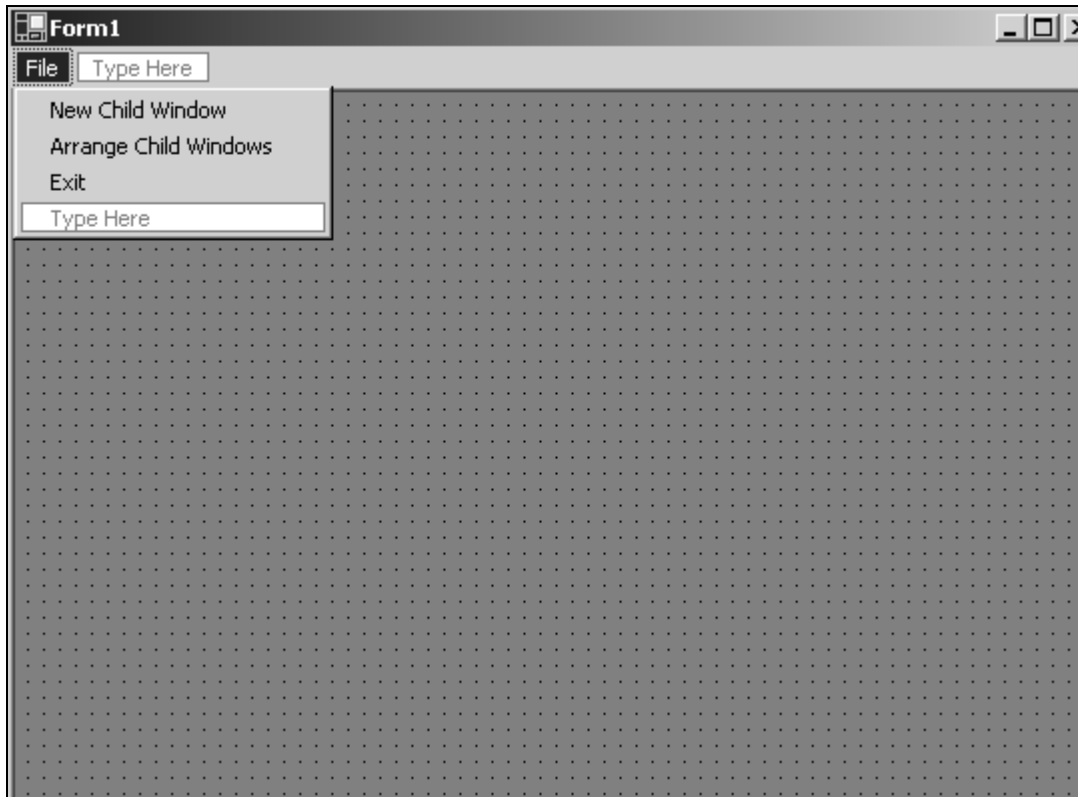
MDI Applications

MDI (Multiple Document Interface) Application is an application in which we can view and work with several documents at once. Example of an MDI application is Microsoft Excel. Excel allows us to work with several documents at once. In contrast, SDI (Single Document Interface) applications are the applications which allows us to work with a single document at once. Example of a single document application is Microsoft Word in which only one document is visible at a time. Visual Basic .NET provides great support for creating and working with MDI applications. In general, MDI applications are mostly used by financial services organizations where the user needs to work with several documents at once.

13.1 Creating MDI Applications

Let's create an MDI application. Open a new Windows Application in Visual Basic .NET. The application will open with a default form, Form1. Add another form, Form2 to this application by right-clicking on the project name in *Solution Explorer* window and selecting *Add->Add Windows Form*. You can add some controls to Form2. For this application we will make Form1 as the *MDI parent window* and Form2 as *MDI child window*. MDI child forms are important for MDI Applications as users interact mostly through child forms. Select Form1 and in its Properties Window under the Windows Style section, set the property *IsMdiContainer* to **True**. Setting it to true designates this form as an MDI container for the child windows. Once you set that property to true the form changes its color. Now, from the toolbox drag a MainMenu component onto Form1. We will display child windows when a menu item is clicked. Name the top-level menu item to File with submenu items as New Child Window, Arrange Child Windows and Exit. The whole form should look like the image below.

Coalesce



What will happen with this application is, a new child window is displayed each time the New Child Window menu item is clicked, all child windows will be arranged when you click Arrange Child Windows menu item. For that, open the code designer window and place the following code.

```
Public Class Form1 Inherits System.Windows.Forms.Form

Dim childForm As Integer = 0
Dim childForms(5) As Form2
'declaring an array to store child windows
'five child windows (Form2) will be displayed

#Region " Windows Form Designer generated code "

Private Sub Form1_Load(ByVal sender As System.Object, ByVal_
As System.EventArgs) Handles MyBase.Load
End Sub

Private Sub MenuItem2_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles MenuItem2.Click
childForm += 1
childForms(childForm) = New Form2()
childForms(childForm).Text = "ChildForm" & Str(childForm)
'setting title for child windows and incrementing the number with an
array
childForms(childForm).MdiParent = Me
childForms(childForm).Show()
End Sub
```

Coalesce

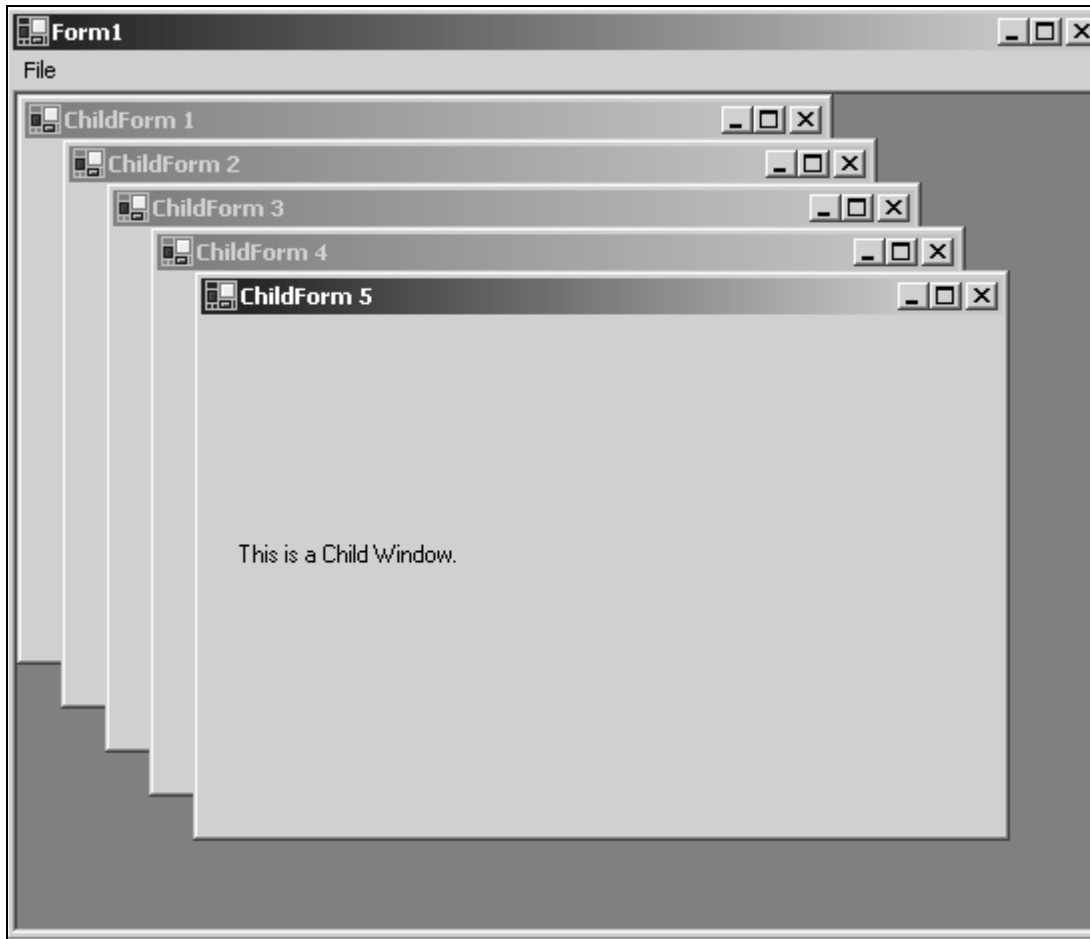
```
Private Sub MenuItem3_Click(ByVal sender As System.Object, _  
ByVal e As System.EventArgs) Handles MenuItem3.Click  
Me.LayoutMdi(MdiLayout.Cascade)  
'arranging child windows on the parent form with predefined_  
'LayoutMdi method  
'Different layouts available are, ArrangeIcons, Cascade _  
'TileHorizontal, TileVertical  
End Sub  
  
Private Sub MenuItem4_Click(ByVal sender As System.Object, _  
ByVal e As System.EventArgs) Handles MenuItem4.Click  
Me.Close()  
'closing the application  
End Sub  
  
End Class
```

When you run the application and click "New Child Window" menu item, a new child window is displayed. Five child windows will be displayed as we declared an array of five in code. The image below displays the output.



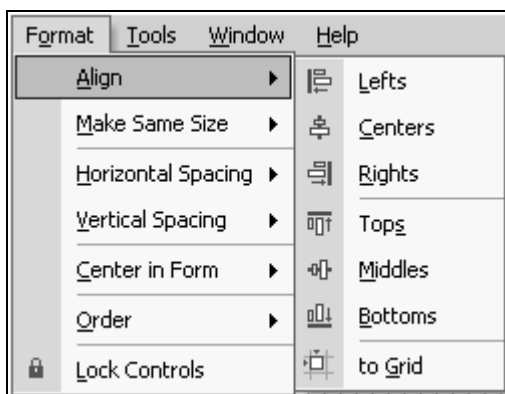
When you click on "Arrange Child Windows" menu item, all child windows are arranged. It looks like the image below.

Coalesce



13.2 Format Menu

The Format Menu available in Visual Studio .NET IDE allows us to align, layer and lock controls on a form. The Format menu provides many options for arranging controls. The Format Menu is shown in the image below.



When using Format menu for arranging controls, we need to select the controls in such a way that the last control selected is the primary control to which other controls are aligned. The primary control has dark size handles and all other controls have light size handles.

Coalesce

The different options that are available under the Format menu are listed below.

Align: Aligns all controls with respect to the primary control

Make Same Size: Resizes multiple controls on a form

Horizontal Spacing: Increases horizontal spacing between controls

Vertical Spacing: Increases vertical spacing between controls

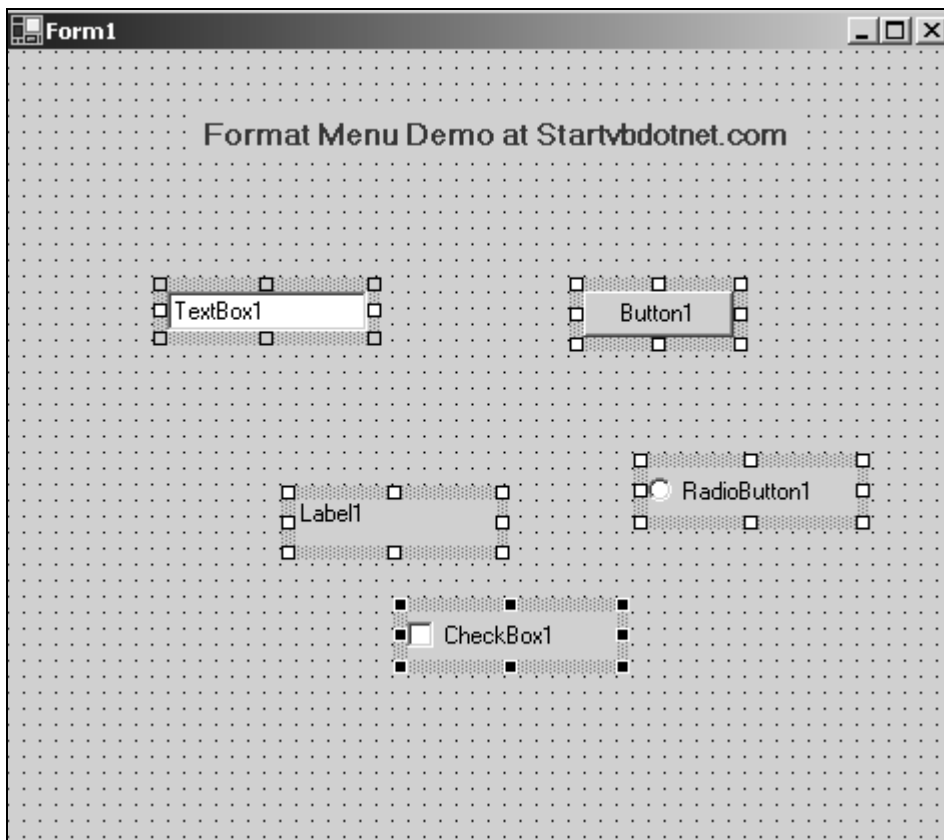
Center in Form: Centers the controls on form

Order: Layers controls on form

Lock Controls: Locks all controls on form

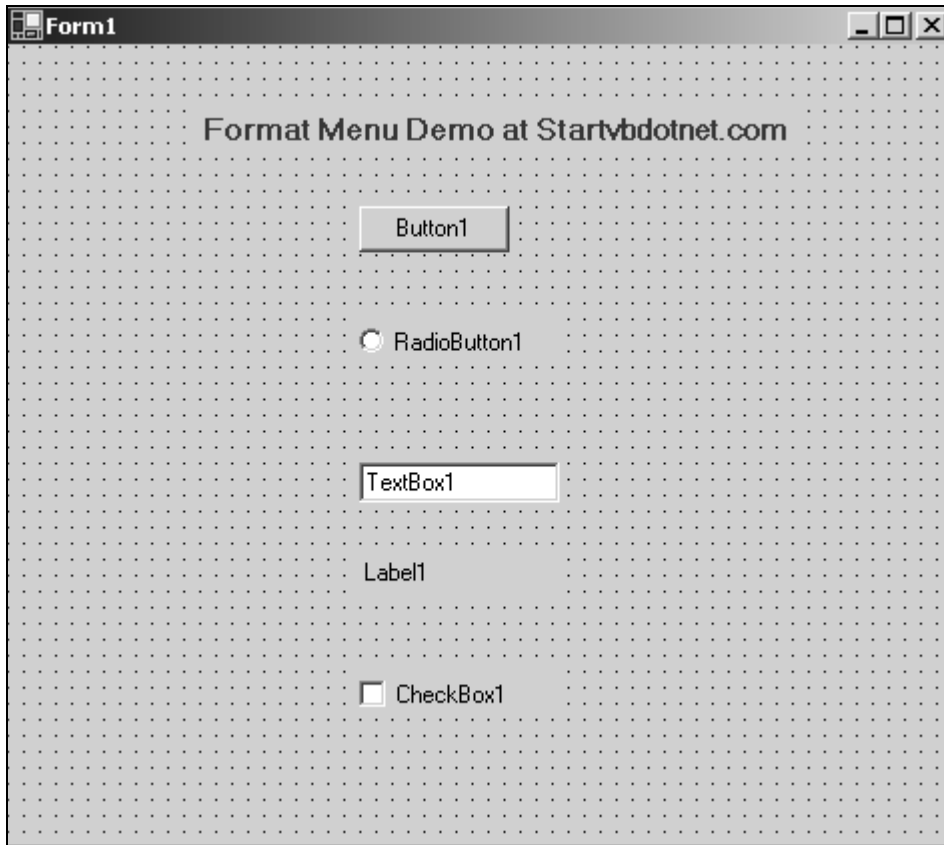
13.3 Aligning multiple controls on a form

Let's work with an example. Open a new form and drag some controls onto it from the toolbox. Select the controls you want to align so that the last control is the primary control to which others are aligned. On the Format menu, point to align and then click any of the seven options available. The image below displays the controls after selection. From the image notice the CheckBox control with dark handles. That's the primary control.



The image below displays controls that are aligned after selecting the option Lefts under Align from the Format menu. All the controls are left-aligned.

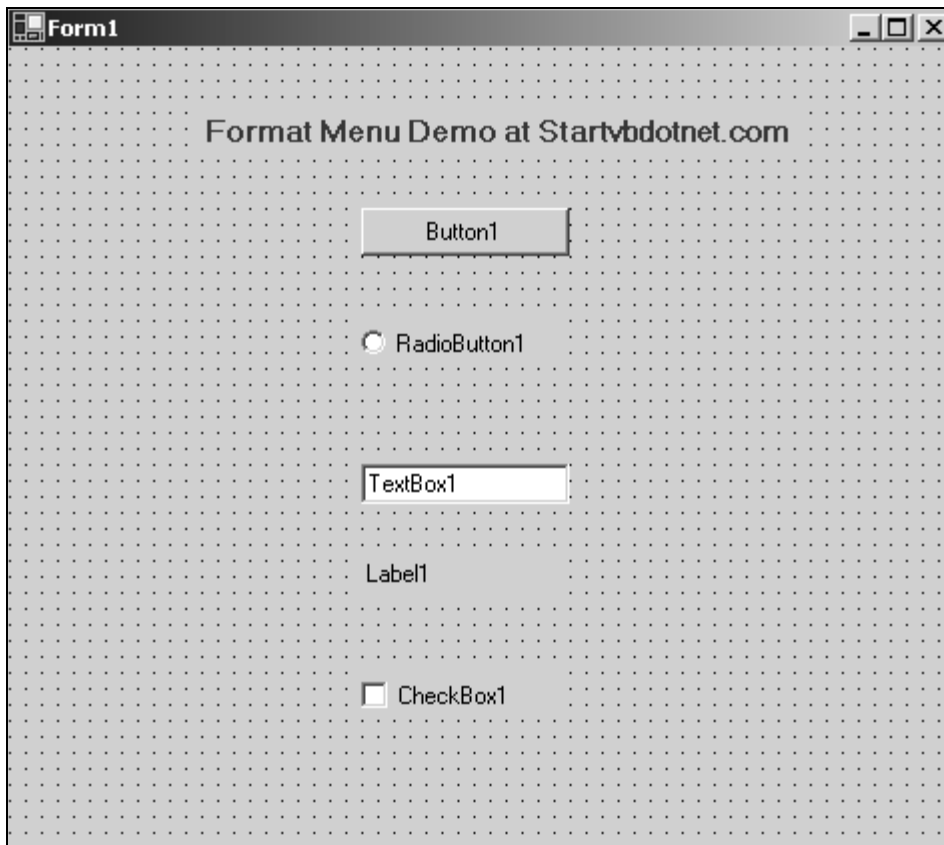
Coalesce



13.4 Make Same Size

The Make Same Size option under Format menu provides four options using which we can make all the controls same in size, width or height. The image below displays controls with the option width that makes all the controls same in width as the primary control.

Coalesce



13.5 Locking Controls

To lock all controls so that they cannot be moved accidentally, select Format menu and click lock controls. The image below displays that.

Coalesce

The image shows a screenshot of a Windows application window titled "Form1". The window has a standard Windows XP-style title bar with minimize, maximize, and close buttons. The main area of the form has a light gray background with a fine grid pattern. At the top center, the text "Format Menu Demo at Startvbdotnet.com" is displayed. Below this text, there are five controls arranged in a layout that suggests a menu structure. On the left is a "Label1" control. To its right is a "TextBox1" control. Below the "Label1" and "TextBox1" are two controls: a "RadioButton1" and a "CheckBox1". At the bottom center of the form is a "Button1" control. All controls have a standard Windows XP visual style with a gray border and a slight shadow.

Coalesce

Chapter 14

Controls

A *control* is an object that can be drawn on to the Form to enable or enhance user interaction with the application. Examples of these controls include the TextBoxes, Buttons, Labels, Radio Buttons etc. All these Windows Controls are based on the *Control* class, which is the base class for all the controls. Visual Basic allows us to work with controls in two ways: at *design time* and at *run time*. Working with controls at design time means the controls are visible to us and we can work with them by dragging and dropping them from the Toolbox and setting its properties in the property window. Working at run time means that the controls are not visible while designing, are created and are assigned properties in code and visible only when the application is set to run. There are many new controls added in Visual Basic.NET and we will be working with some of the most popular controls in this section.

14 Windows ToolBox

In this session, we are going to see some of the commonly used controls which are listed below

- Button Control
- TextBox Control
- RichTextBox Control
- Label Control
- LinkLabel Control
- CheckBox Control
- RadioButton Control
- ListBox Control
- CheckedListBox Control
- ComboBox Control
- Panel
- GroupBox
- PictureBox
- Date TimePicker
- MonthCalendar
- ToolTip
- Splitter
- Menus
- Dialog Boxes

14.1 Properties windows

Notable properties of most of the above Windows Controls which are based on the Control class itself are summarized in the tables below. You can always find the properties of the controls by pressing *F4* on the keyboard or by selecting from *View->Properties Window* from the main menu. The properties are divided into sections below as visible in the properties window.

Coalesce

Properties/Attributes	Description
BackColor	Gets/Sets the background color for the control
BackgroundImage	Get/Sets the background image in the control
BorderStyle	Gets/Sets the border type of the control
Cursor	Gets/Sets the cursor to be displayed when the user moves the mouse over the control
FlatStyle	Gets/Sets the flat style of the control
Font	Gets/Sets the font for the control
ForeColor	Gets/Sets the foreground color for the control
Image	Gets/Sets the image to be displayed on the control
ImageAlign	Gets/Sets the alignment of an image that is displayed in the control
ImageIndex	Gets/Sets the image list index value of the image displayed in the control
ImageList	Gets/Sets the ImageList that contains the images displayed in a control
RightToLeft	Gets/Sets the value indicating if the alignment of the control's elements is reversed to support right-to-left fonts
ScrollBars	Gets/Sets the kind of scrollbars to display in the control
Text	Gets/Sets the text to be displayed for the control
TextAlign	Gets/Sets the alignment of text in the control

Properties/Attributes	Description
AcceptsReturn	Gets or sets a value indicating whether pressing ENTER in a control creates a new line of text in the control or activates the default button for the form
AcceptsTab	Gets/Sets whether the control accepts Tab or not
AllowDrop	Gets/Sets a value which specifies if the control can accept data dropped into it
AutoSize	Gets/Sets a value specifying whether a control is automatically resized to fit its contents
ContextMenu	Gets/Sets the shortcut menu associated with the control
Enabled	Gets/Sets a value which specifying if the control is enabled
HorizontalScrollBar	Gets/Sets a HorizontalScrollBar for a Control
ImeMode	Gets/Sets the state of an Input Method Editor
MultiColumn	Sets the control's text to be displayed in multiple columns
MultiLine	Sets the control to accept multiple lines of text
ReadOnly	Sets the control to be a ReadOnly Control
ScrollAlways Visible	Sets a ScrollBar for the control
Sorted	Sorts the data items in a control

Coalesce

TabIndex	Gets/Sets the tab order of the control in its container
TabStop	Gets/Sets a value specifying if the user can tab to the control with the Tab key
TextAlign	Gets/Sets the alignment of text on a control
Visible	Gets/Sets a value specifying if the control is visible

Properties/Attributes	Description
DataBindings	Gets the data bindings for the control
Tag	Gets/Sets an object that contains data about the control
Properties/Attributes	Description
Locked	Gets/Sets whether the control is Locked or not
Modifiers	Sets the Modifier for control
Name	Gets/Sets name for the Control
Properties/Attributes	Description
Anchor	Gets/Sets which edges of the control can be anchored
Location	Gets/Sets the co-ordinates of the upper-left corner of the control
Size	Gets/Sets the height and width of the control

Button control

One of the most popular control in Visual Basic is the Button Control (previously Command Control) along with TextBoxes. They are the controls which we click and release to perform some action. Buttons are used mostly for handling events in code, say for sending data entered in the form to the database and so on. The default event of the Button is the Click event and the Button class is based on the *ButtonBase* class which in turn is based on the Control class.

Event of the Button

The default event of the Button is the *Click* event. When a Button is clicked, it responds with the Click Event. The Click event of Button looks like this in code:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal  
e As_  
System.EventArgs) Handles Button1.Click  
'You place the code here to perform action when Button is clicked  
End Sub
```

Working with Buttons

Well, it's time to work with Buttons. Double-click on the Button in the toolbox. Button gets added to the Form. The default text on the Button is Button1. Click on Button1 and select its properties by pressing *F4* on the keyboard or by selecting *View->Properties Window* from the main menu. Doing so, displays the Properties window for Button1.

Coalesce

Important Properties of Button1 from Properties Window:

Appearance

Appearance section of the properties window allows us to make changes to the appearance of the Button. With the help of *BackColor* and *Background Image* properties we can set a background color and a background image to the button. We set the font color and font style for the text that appears on button with *ForeColor* and the *Font* property. We change the appearance style of the button with the *FlatStyle* property. We can change the text that appears on button with the *Text* property and with the *TextAlign* property we can set where on the button the text should appear from a predefined set of options.

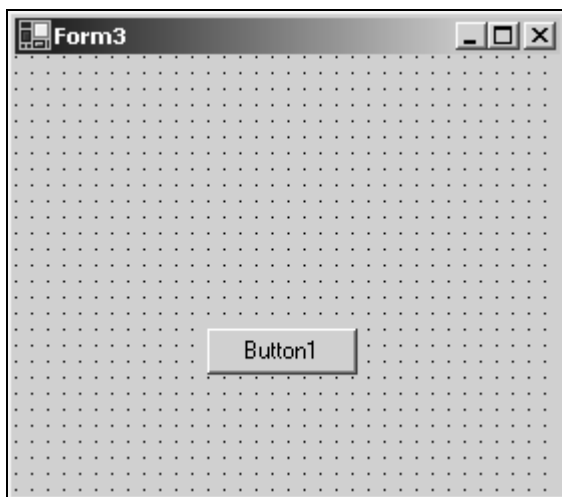
Behavior

Notable Behavior properties of the Button are the *Enabled* and *Visible* properties. The Enabled property is set to *True* by default which makes the button enabled and setting it's property to False makes the button Disabled. With the Visible property we can make the Button Visible or Invisible. The default value is set to *True* and to make the button Invisible set it's property to False.

Layout

Layout properties are about the look of the Button. Note the *Dock* property here. A control can be docked to one edge of its parent container or can be docked to all edges and fill the parent container. The default value is set to none. If you want to dock the control towards the left, right, top, bottom and center you can do that by selecting from the button like image this property displays. With the *Location* property you can change the location of the button. With the *Size* property you can set the size of the button. Apart from the Dock property you can set it's size and location by moving and stretching the Button on the form itself.

Below is an image of the button control. You can view notable properties of this Button control which it shares with other controls by *clicking here*.



Coalesce

Creating a Button in Code:

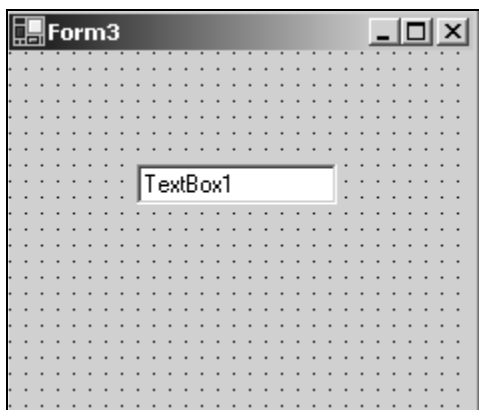
Code to create (that is run time) a Button looks like this:

```
Public Class Form1 Inherits System.Windows.Forms.Form
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
Dim Button1 as New Button()
'declaring the button,Button1
Button1.Text="Creating a Button"
'setting the text to be displayed on the Button
Button1.Location=New Point(100,50)
'setting the location for the Button where it should be created
Button1.Size=New Size(75,23)
'setting the size of the Button
Me.Controls.Add(Button1)
'adding the Button that is created
'the Me keyword is used to refer to the current object, in this case the
Form
End Sub
End Class
```

TextBox Control

Everyone working with Windows should be familiar with textboxes. This control looks like a box and accepts input from the user. The TextBox is based on the *TextBoxBase* class which in turn is based on the control class. TextBoxes are used to accept input from the user or used to display text. By default we can enter up to 2048 characters in a TextBox but if the Multiline property is set to True we can enter up to 32KB of text.

The image below is a Textbox. You can view the properties of the TextBox which are also shared by other controls by *clicking here*.



Some Notable Properties:

Some important properties in the *Behavior* section of the Properties Window for TextBoxes.

Enabled: Default value is True. To disable it set the property to False.

Coalesce

Multiline: Setting this property to True makes the TextBox multiline, which allows to accept multiple lines of text. Default value is False.

PasswordChar: Used to set the password character. The text displayed in the TextBox will be the character set by the user. Say if you enter * in that property, the text that is entered in the TextBox is displayed as *.

ReadOnly: Makes this TextBox readonly, which doesn't allow to enter any text.

Visible: Default value is True to hide it set the property to False.

Important properties in the Appearance section

TextAlign: Allows to align the text from three possible options. The default value is left and you can set the alignment of text to right or center from the list.

Scrollbars: Allows to add a scrollbar to a Textbox. Very useful when the TextBox is multiline. You have four options with this property which are None, Horizontal, Vertical and Both. Depending on the size of the TextBox any one of those can be used.

The TextBox Event

The default event of the TextBox is the *TextChanged* Event which looks like this in code:

```
Private Sub TextBox1_TextChanged(ByVal sender As System.Object,
ByVal e As _
System.EventArgs) Handles TextBox1.TextChanged
End Sub
```

Working With TextBoxes

Lets go with some examples to work with TextBoxes.

Drag two TextBoxes (TextBox1,TextBox2) and a Button (Button1) from the toolbox.

Code to Display some text in the TextBox

We want to display some text, say "Welcome to TextBox", in TextBox1 when the Button is clicked. Place this code in the click event of the Button.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles_ Button1.Click
TextBox1.Text = "Welcome to TextBox"
End Sub
```

Code to Work with PassWord Character

Set the PasswordChar property of TextBox2 to *. Setting that will make the text entered in TextBox2 to be displayed as *. We want to display what is entered in TextBox2 in TextBox1. The code for that looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As _ System.EventArgs) Handles Button1.Click
TextBox1.Text = TextBox2.Text
```

Coalesce

```
End Sub
```

When you run the program and enter some text in TextBox2, text will be displayed as *. When you click the Button, the text you entered in TeXtBox2 will be displayed as plain text in TextBox1.

Code to control input in a TextBox.

We can make sure that the TextBox can accept only characters or numbers which can restrict accidental operations. For example, adding two numbers, 100+1A9 cannot return anything. To avoid that sort of operations we can do with the KeyPress event of the TextBox.

Code that allows you to enter only double digits in a TextBox looks like this:

```
Private Sub TextBox1_KeyPress(ByVal sender As Object, ByVal e As  
_ System.Windows.Forms.KeyPressEventArgs) Handles  
TextBox1.KeyPress  
If(e.KeyChar < "10" Or e.KeyChar > "100") Then  
MessageBox.Show("Enter Double Digits")  
End If  
End Sub
```

Creating a TextBox in Code

Code to create a TextBox looks like this:

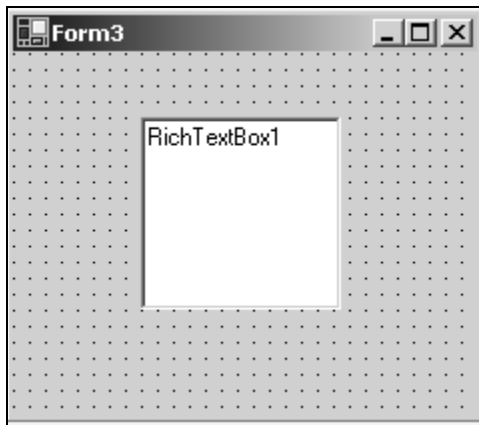
```
Public Class Form1 Inherits System.Windows.Forms.Form  
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As  
System.EventArgs)_  
Handles MyBase.Load  
Dim TextBox1 as New TextBox()  
TextBox1.Text="Hello Mate"  
TextBox1.Location=New Point(100,50)  
TextBox1.Size=New Size(75,23)  
Me.Controls.Add(TextBox1)  
End Sub  
End Class
```

RichTextBox

RichTextBoxes are similar to TextBoxes but they provide some advanced features over the standard TextBox. RichTextBox allows formatting the text, say adding colors, displaying particular font types and so on. The RichTextBox like the TextBox is based on the *TextBoxBase* class which is based on the control class. These RichTextBoxes came into existence because many word processors these days allow us to save text in a rich text format. With RichtextBoxes we can also create our own word processors. We have two options when accessing text in a RichTextBox, text and rtf (rich text format). Text holds text in normal text and rtf holds text in a rich text format.

Below is the image of a RichTextBox. You can view the properties for the RichTextBox which are also shared by other controls by *clicking here*.

Coalesce



The Event for RichTextBox

The default event associated with the RichTextBox is the TextChanged which looks like this in code:

```
Private Sub RichTextBox2_TextChanged(ByVal sender As
System.Object, _
ByVal e As System.EventArgs) Handles RichTextBox2.TextChanged
End Sub
```

Working with Examples

Code for creating bold and italic text in a RichTextBox

Drag a RichTextBox (RichTextBox1) and a Button (Button1) onto the form. Enter some text in RichTextBox1, say, "We are working with RichTextBoxes". Write the following code in the click event of Button1. This code will search for the text we mention and sets it to be displayed as Bold font or Italic font based on what text is searched for. The code looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e_
As System.EventArgs) Handles Button1.Click
RichTextBox1.SelectionStart = RichTextBox1.Find("are")
'using the Find method to find the text "are" and setting it's_
'return property to SelectionStart which selects the text to format
Dim ifont As New Font(RichTextBox1.Font, FontStyle.Italic)
'creating a new font object to set the font style
RichTextBox1.SelectionFont = ifont
'assigning the value selected from the RichTextBox the font style
RichTextBox1.SelectionStart = RichTextBox1.Find("working")
Dim bfont As New Font(RichTextBox1.Font, FontStyle.Bold)
RichTextBox1.SelectionFont = bfont
End Sub
```

When you run the above said code and click Button1, the text "are" is displayed in Italic and the text "working" is displayed in Bold font. The image below displays the output.

Code for Setting the Color of Text

Coalesce

Lets work with previous example. Code for setting the color for particular text looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e  
As _  
System.EventArgs) Handles Button1.Click  
RichTextBox1.SelectionStart = RichTextBox1.Find("are")  
'using the Find method to find the text "are" and setting it's return _  
'property to SelectionStart which selects the text  
RichTextBox1.SelectionColor = Color.Blue  
'setting the color for the selected text with SelectionColor property  
RichTextBox1.SelectionStart = RichTextBox1.Find("working")  
RichTextBox1.SelectionColor = Color.Yellow  
End Sub
```

The output when the Button is Clicked is the text "are" being displayed in Blue and the text "working" in yellow as shown in the image below.

The screenshot shows a Windows Form titled "Form1" with a light gray background. It contains three distinct sections for demonstrating RichTextBox capabilities:

- Bold and Italic Demo:** Located in the top left, it features a RichTextBox with the text "We *are* **working** with RichTextBoxes". Below the text box is a button labeled "Bold and Italic".
- Font Color Demo:** Located in the top right, it features a RichTextBox with the text "We are working with RichTextBoxes". Below the text box is a button labeled "Font Color".
- RTF Demo:** Located at the bottom center, it consists of two RichTextBoxes, each containing the text "Saving a file to rtf". Below these text boxes are two buttons: "Save" and "Load".

Coalesce

Code for Saving Files to RTF

Drag two RichTextBoxes and two Buttons (Save, Load) onto the form. When you enter some text in RichTextBox1 and click on Save button, the text from RichTextBox1 is saved into a rtf (rich text format) file and when you click on Load button, the text from the rtf file is displayed into RichTextBox2. The code for that looks like this:

```
Private Sub Save_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs)_
Handles Save.Click
RichTextBox1.SaveFile("hello.rtf")
'using SaveFile method to save text in a rich text box to hard disk
End Sub
Private Sub Load_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs)_
Handles Load.Click
RichTextBox2.LoadFile("hello.rtf")
'using LoadFile method to read the saved file
End Sub

End Class
```

The files which we create using the SaveFile method are saved in the bin directory of the Windows Application. You can view output of the above said code in the image above.

Label

Labels are those controls which are used to display text in other parts of the application. They are based on the *control* class.

Notable property of the label control is the *text* property which is used to set the text for the label.

Click here for all other properties of the Label which also are shared by other controls.

The Event of Label

The default event associated with label is the *Click* event which looks like this in code:

```
Private Sub Label1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles_ Label1.Click
End Sub
```

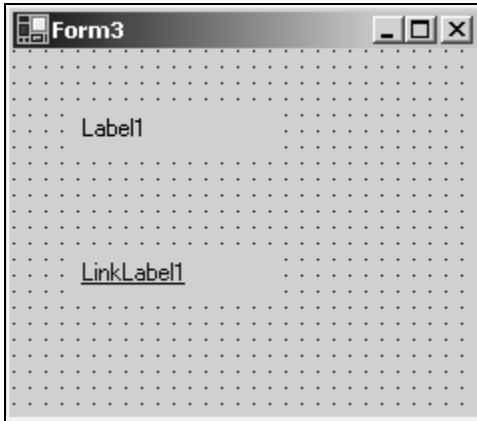
Creating a Label in Code

Code for creating a Label looks like this:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles_ MyBase.Load
Dim Label1 As New Label()
Label1.Text = "Label"
```

Coalesce

```
Label1.Location = New Point(135, 70)
Label1.Size = New Size(30, 30)
Me.Controls.Add(Label1)
End Sub
```



Link Label

LinkLabel is similar to a Label control but the difference is, they display a hyperlink. Even multiple hyperlinks can be specified in the text of the control and each hyperlink can perform a different task within the application. They are based on the *Label* class which in turn are based on the control class.

Notable properties of the LinkLabel control are the *ActiveLinkColor*, *LinkColor* and *LinkVisited* which are used to set the link color.

Click here for all other properties of the LinkLabel which are also shared by other controls.

Event of the LinkLabel

The default event associated with LinkLabel is the *LinkClicked* event which looks like this in code:

```
Private Sub LinkLabel1_LinkClicked(ByVal sender As System.Object,
_
ByVal e As System.Windows.Forms.LinkLabelLinkClickedEventArgs)
Handles LinkLabel1.LinkClicked
End Sub
```

Working with LinkLabel

Drag a LinkLabel (LinkLabel1) onto the form. When we click this LinkLabel it's going to connect us to the website "www.startvbdotnet.com". Double-click on LinkLabel1 and place this code in it's event which looks like this:

```
Private Sub LinkLabel1_LinkClicked(ByVal sender As System.Object,
ByVal _
e As System.Windows.Forms.LinkLabelLinkClickedEventArgs)
Handles LinkLabel1.LinkClicked
System.Diagnostics.Process.Start("www.startvbdotnet.com")
End Sub
```

Coalesce

```
'using the start method of system.diagnostics.process class, process  
class gives access to  
'local and remote processes  
End Sub
```

Creating a LinkLabel in Code

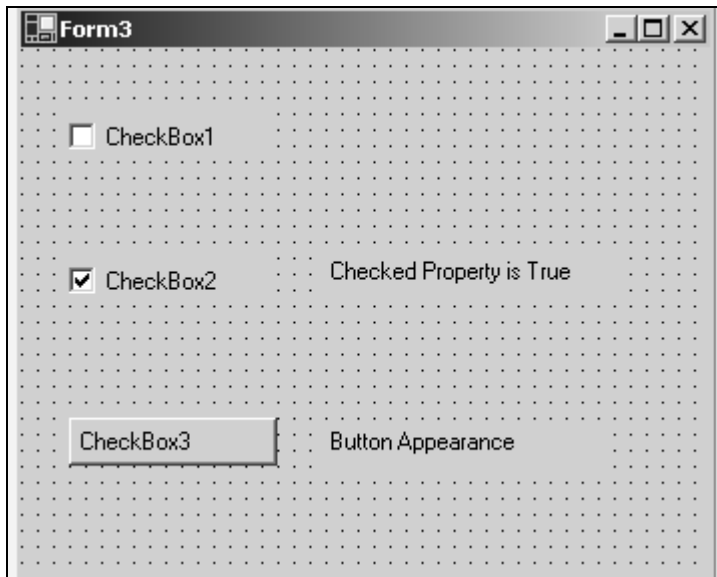
Code for creating a LinkLabel looks like this:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles MyBase.Load  
Dim LinkLabel1 As New LinkLabel()  
LinkLabel1.Text = "Label"  
LinkLabel1.Location = New Point(135, 70)  
LinkLabel1.Size = New Size(30, 30)  
Me.Controls.Add(LinkLabel1)  
End Sub
```

CheckBox

CheckBoxes are the controls which gives us an option to select, say yes/no or true/false. A checkbox is clicked to select and clicked again to deselect some option. When a checkbox is selected, a check (a tick mark) appears indicating a selection. The CheckBox control is based on the *TextBoxBase* class which is based on the Control class.

Below is the image of a CheckBox. The properties of the checkboxes which are also shared by other controls can be viewed by *clicking here*.



Coalesce

Notable Properties

Important properties of the CheckBox in the *Appearance* section of the properties window are:

Appearance: Default value is Normal. Set the value to Button if you want the CheckBox to be displayed as a Button.

BackgroundImage: Used to set a background image for the checkbox.

CheckAlign: Used to set the alignment for the CheckBox from a predefined list.

Checked: Default value is False, set it to True if you want the CheckBox to be displayed as checked.

CheckState: Default value is Unchecked. Set it to True if you want a check to appear. When set to Indeterminate it displays a check in gray background.

FlatStyle: Default value is Standard. Select the value from a predefined list to set the style of the checkbox.

Important property in the Behavior section of the properties window is the *ThreeState* property which is set to False by default. Set it to True to specify if the Checkbox can allow three check states than two.

Event of the CheckBox

The default event of the CheckBox is the *CheckedChange* event which looks like this in code:

```
Private Sub CheckBox1_CheckedChanged(ByVal sender As  
System.Object, _  
ByVal e As System.EventArgs) Handles CheckBox1.CheckedChanged  
End Sub
```

Working with CheckBoxes

Lets work with an example. Drag a CheckBox (CheckBox1), TextBox (TextBox1) and a Button (Button1) from the Toolbox.

Code to display some text when the Checkbox is checked looks like this:

```
Private Sub CheckBox1_CheckedChanged(ByVal sender As  
System.Object, _  
ByVal e As System.EventArgs) Handles CheckBox1.CheckedChanged  
TextBox1.Text = "CheckBox Checked"  
End Sub
```

Code to check a CheckBox's state looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
ByVal e As System.EventArgs) Handles Button3.Click  
If CheckBox1.Checked = True Then  
TextBox1.Text = "Checked"  
Else  
TextBox1.Text = "UnChecked"  
End If  
End Sub
```

Coalesce

Creating a CheckBox in Code

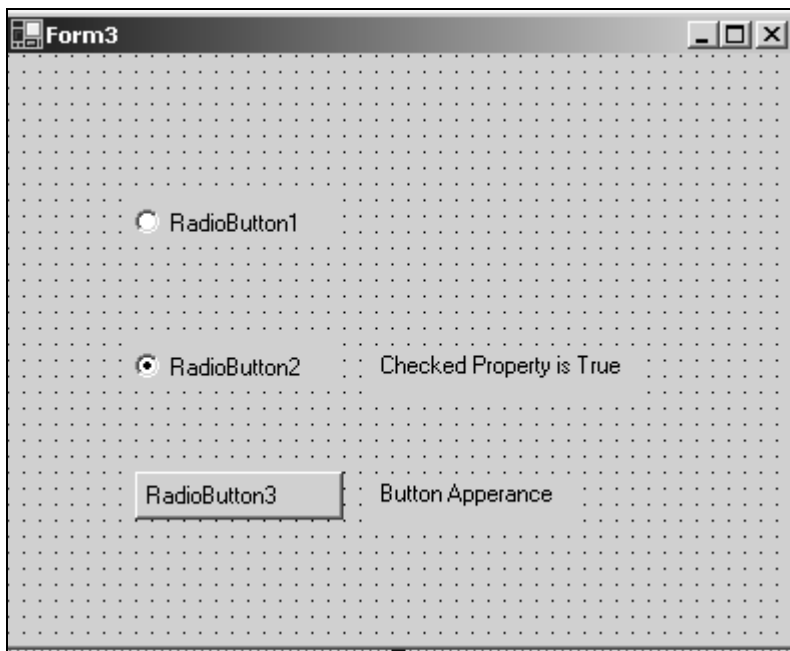
The code to create a CheckBox looks like this:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles MyBase.Load  
Dim CheckBox1 As New CheckBox()  
CheckBox1.Text = "Checkbox1"  
CheckBox1.Location = New Point(100, 50)  
CheckBox1.Size = New Size(95, 45)  
Me.Controls.Add(CheckBox1)  
End Sub
```

RadioButton

RadioButtons are similar to CheckBoxes but RadioButtons are displayed as rounded instead of box, as with a checkbox. Like CheckBoxes, RadioButtons are used to select and deselect options but they allow us to choose from mutually exclusive options. The RadioButton control is based on the *ButtonBase* class which is based on the *Control* class. A major difference between CheckBoxes and RadioButtons is, RadioButtons are mostly used together in a group.

Below is an image of a RadioButton. The properties of the RadioButton which are also shared by other controls can be viewed by *clicking here*.



Important properties of the RadioButton in the *Appearance* section of the properties window are:

Appearance: Default value is Normal. Set the value to Button if you want the radioButton to be displayed as a Button.

Coalesce

BackgroundImage: Used to set a background image for the RadioButton.

CheckAlign: Used to set the alignment for the RadioButton from a predefined list.

Checked: Default value is False, set it to True if you want the RadioButton to be displayed as checked.

FlatStyle: Default value is Standard. Select the value from a predefined list to set the style of the RadioButton.

Event of the RadioButton

The default event of the RadioButton is the *CheckedChange* event which looks like this in code:

```
Private Sub RadioButton1_CheckedChanged(ByVal sender As
System.Object, _
ByVal e As System.EventArgs) Handles
RadioButton1.CheckedChanged
End Sub
```

Working with Examples

Drag a RadioButton (RadioButton1), TextBox (TextBox1) and a Button (Button1) from the Toolbox.

Code to display some text when the RadioButton is selected looks like this:

```
Private Sub RadioButton1_CheckedChanged(ByVal sender As
System.Object, _
ByVal e As System.EventArgs) Handles
RadioButton1.CheckedChanged
TextBox1.Text = "RadioButton Selected"
End Sub
```

Code to check a RadioButton's state looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click
If RadioButton1.Checked = True Then
TextBox1.Text = "Selected"
Else
TextBox1.Text = "Not Selected"
End If
End Sub
```

Creating a RadioButton in Code

Code to create a RadioButton looks like this:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
Dim RadioButton1 As New RadioButton()
RadioButton1.Text = "RadioButton1"
RadioButton1.Location = New Point(120,60)
RadioButton1.Size = New Size(100, 50)
```

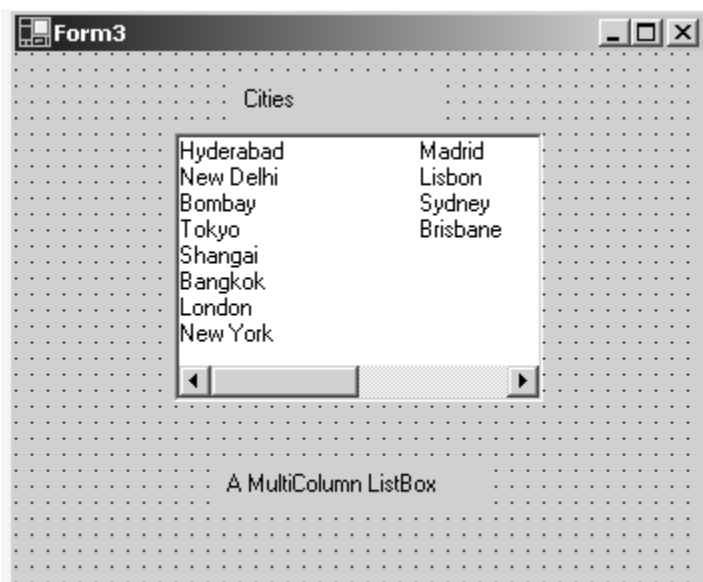
Coalesce

```
Me.Controls.Add(RadioButton1)
End Sub
```

ListBox

The ListBox control displays a list of items from which we can make a selection by clicking. We can select one or more items from the list of items displayed. The ListBox control is based on the *ListControl* class which in turn is based on the Control class.

The image below displays a ListBox. You can view the properties of a ListBox which are also shared by other controls by *clicking here*.



Notable Properties of the ListBox

In the **Behavior** Section

HorizontalScrollbar: Displays a horizontal scrollbar to the ListBox. Works when the ListBox has MultipleColumns.

MultiColumn: The default value is set to False. Set it to True if you want the list box to display multiple columns.

ScrollAlwaysVisible: Default value is set to False. Setting it to True will display both Vertical and Horizontal scrollbar always.

SelectionMode: Default value is set to none. Select option One if you want only one item to be selected at a time. Select it to MultiSimple if you want multiple items to be selected. Setting it to MultiExtended allows you to select multiple items with the help of Shift, Control and arrow keys on the keyboard.

Sorted: Default value is set to False. Set it to True if you want the items displayed in the ListBox to be sorted by alphabetical order.

Coalesce

In the **Data Section**

Notable property in the Data section of the Properties window is the *Items* property. The Items property allows us to add the items we want to be displayed in the list box. Doing so is simple, click on the ellipses to open the String Collection Editor window and start entering what you want to be displayed in the ListBox. After entering the items click OK and doing that adds all the items to the ListBox.

The **ListBox** Event

The default event associated with the ListBox is the *SelectedIndexChanged* which looks like this in code:

```
Private Sub ListBox1_SelectedIndexChanged(ByVal sender As  
System.Object, _  
ByVal e As System.EventArgs) Handles  
ListBox1.SelectedIndexChanged  
End Sub
```

Working with **ListBoxes**

Drag a TextBox and a ListBox control to the form and add some items to the ListBox with it's items property.

Referring to Items in the ListBox

Items in a ListBox are referred by index. When items are added to the ListBox they are assigned an index. The first item in the ListBox always has an index of 0 the next 1 and so on.

Code to display the index of an item

```
Private Sub ListBox1_SelectedIndexChanged(ByVal sender As  
System.Object, _  
ByVal e As System.EventArgs) Handles  
ListBox1.SelectedIndexChanged  
TextBox1.Text = ListBox1.SelectedIndex  
'using the selected index property of the list box to select the index  
End Sub
```

When you run the code and select an item in the ListBox, it's index is displayed in the textbox.

Code to count the number of Items in a ListBox

Add a Button to the form and place the following code in the click event of the Button:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e _  
As System.EventArgs) Handles Button1.Click  
TextBox1.Text = ListBox1.Items.Count  
'counting the number of items in the ListBox with the Items.Count  
End Sub
```


Coalesce

When you run the program and click the Button, it will display the number of items available in the ListBox.

Code to display the item selected from ListBox in a TextBox.

```
Private Sub ListBox1_SelectedIndexChanged(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles  
    ListBox1.SelectedIndexChanged  
    TextBox1.Text = ListBox1.SelectedItem  
    'using the selected item property  
End Sub
```

When you run the program and click an item in the ListBox that item will be displayed in the TextBox.

Code to Remove item from a list box

You can remove all items or one particular item from the list box.

Code to remove a particular item.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e _  
    As System.EventArgs) Handles Button1.Click  
    ListBox1.Items.RemoveAt(4)  
    'removing an item by specifying it's index  
End Sub
```

Code to remove all items from the list box.

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Button1.Click  
    ListBox1.Items.Clear()  
    'using the clear method to clear the list box  
End Sub
```

CheckedListBox

As the name says, CheckedListBox is a combination of a ListBox and a CheckBox. It displays a ListBox with a CheckBox towards it's left. The CheckedListBox class is derived from the *ListBox* class and is based on that class. Since the CheckedListBox is derived from the ListBox, it shares all the members of ListBox .

You can view the properties of CheckedListBox which are also shared by other controls by *clicking here*.

Coalesce

Notable Properties of CheckedListBox

The notable property in the appearance section of the properties window is the *ThreeDCheckBoxes* property which is set to False by default. Setting it to True makes the CheckedListBox to be displayed in Flat or Normal style.

Notable property in the behavior section is the *CheckOnClick* property which is set to False by default. When set to False it means that to check or uncheck an item in the CheckedListBox we need to double-click the item. Setting it to True makes an item in the CheckedListBox to be checked or unchecked with a single click.

Notable property in the Data section is the *Items* property with which we add items to the CheckedListBox.

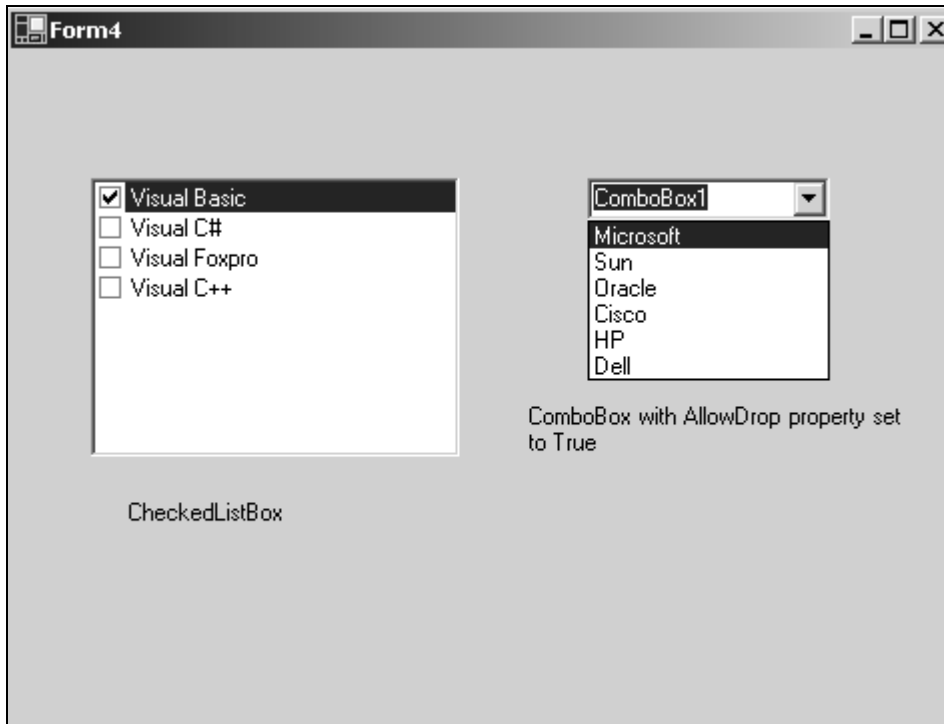
Event of CheckedListBox

The Default Event associated with the CheckedListBox is *SelectedIndexChanged* which looks like this in code:

```
Private Sub CheckedListBox1_SelectedIndexChanged(ByVal sender  
As System.Object, _  
ByVal e As System.EventArgs) Handles  
CheckedListBox1.SelectedIndexChanged  
End Sub
```

Working with CheckedListBoxes is similar to working with ListBoxes. Please refer to examples in that section.

Coalesce



ComboBox

ComboBox is a combination of a TextBox and a ListBox. The ComboBox displays an editing field (TextBox) combined with a ListBox allowing us to select from the list or to enter new text. ComboBox displays data in a drop-down style format. The ComboBox class is derived from the *ListBox* class.

The properties of this control which are also shared by other controls can be viewed by [clicking here](#).

Notable properties of the ComboBox

The *DropDownStyle* property in the Appearance section of the properties window allows us to set the look of the ComboBox. The default value is set to *DropDown*, which means that the ComboBox displays the Text set by its Text property in the Textbox and displays its items in the *DropDownList* below. Setting it to *simple* makes the ComboBox to be displayed with a TextBox and a list box which doesn't drop down.

Setting it to *DropDownList* makes the ComboBox to make selection only from the drop down list and restricts you from entering any text in the textbox.

We can sort the ComboBox with its *Sorted* property which is set to False by Default.

We can add items to the ComboBox with its *Items* property.

Coalesce

The ComboBox Event

The default event associated with the ComboBox is *SelectedIndexChanged* which looks like this in code:

```
Private Sub ComboBox1_SelectedIndexChanged(ByVal sender As
System.Object, ByVal e _
As System.EventArgs) Handles ComboBox1.SelectedIndexChanged
End Sub
```

Examples on working with ComboBoxes

Drag a ComboBox and a TextBox control onto the form. We want to display the selection we make in the ComboBox in the TextBox. The code for that looks like this:

```
Private Sub ComboBox1_SelectedIndexChanged(ByVal sender As
System.Object, ByVal e As _
System.EventArgs) Handles ComboBox1.SelectedIndexChanged
TextBox1.Text = ComboBox1.SelectedItem
'selecting the item from the ComboBox with selected item property
End Sub
```

Removing items from a ComboBox

You can remove all items or one particular item from the list box part of the ComboBox. Code to remove a particular item by its Index number looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e _
As System.EventArgs) Handles Button1.Click
ComboBox1.Items.RemoveAt(4)
'removing an item by specifying its index
End Sub
```

Code to remove all items from the combo box.

```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click
ComboBox1.Items.Clear()
'using the clear method to clear the list box
End Sub
```

CheckedListBox

As the name says, CheckedListBox is a combination of a ListBox and a CheckBox. It displays a ListBox with a CheckBox towards its left. The CheckedListBox class is derived from the *ListBox* class and is based on that class. Since the CheckedListBox is derived from the ListBox, it shares all the members of ListBox .

You can view the properties of CheckedListBox which are also shared by other controls by *clicking here*.

Coalesce

Notable Properties of CheckedListBox

The notable property in the appearance section of the properties window is the *ThreeDCheckBoxes* property which is set to False by default. Setting it to True makes the CheckedListBox to be displayed in Flat or Normal style.

Notable property in the behavior section is the *CheckOnClick* property which is set to False by default. When set to False it means that to check or uncheck an item in the CheckedListBox we need to double-click the item. Setting it to True makes an item in the CheckedListBox to be checked or unchecked with a single click.

Notable property in the Data section is the *Items* property with which we add items to the CheckedListBox.

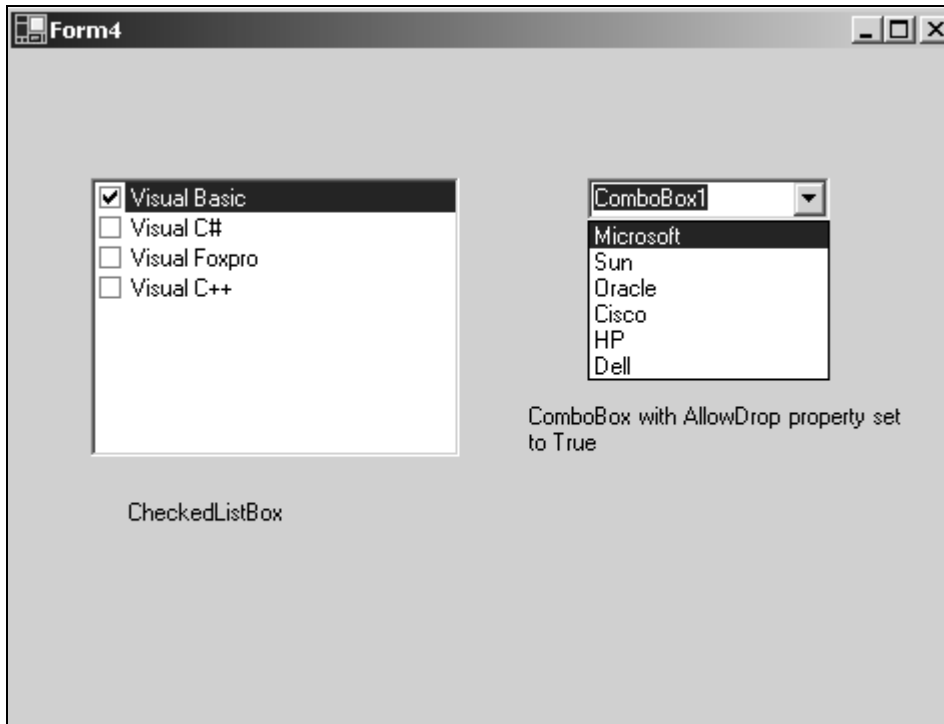
Event of CheckedListBox

The Default Event associated with the CheckedListBox is *SelectedIndexChanged* which looks like this in code:

```
Private Sub CheckedListBox1_SelectedIndexChanged(ByVal sender  
As System.Object, _  
ByVal e As System.EventArgs) Handles  
CheckedListBox1.SelectedIndexChanged  
End Sub
```

Working with CheckedListBoxes is similar to working with ListBoxes. Please refer to examples in that section.

Coalesce



ComboBox

ComboBox is a combination of a TextBox and a ListBox. The ComboBox displays an editing field (TextBox) combined with a ListBox allowing us to select from the list or to enter new text. ComboBox displays data in a drop-down style format. The ComboBox class is derived from the *ListBox* class.

The properties of this control which are also shared by other controls can be viewed by [clicking here](#).

Notable properties of the ComboBox

The *DropDownStyle* property in the Appearance section of the properties window allows us to set the look of the ComboBox. The default value is set to *DropDown*, which means that the ComboBox displays the Text set by its Text property in the Textbox and displays its items in the *DropDownList* below. Setting it to *simple* makes the ComboBox to be displayed with a TextBox and a list box which doesn't drop down.

Setting it to *DropDownList* makes the ComboBox to make selection only from the drop down list and restricts you from entering any text in the textbox.

We can sort the ComboBox with its *Sorted* property which is set to False by Default.

We can add items to the ComboBox with its *Items* property.

Coalesce

The ComboBox Event

The default event associated with the ComboBox is *SelectedIndexChanged* which looks like this in code:

```
Private Sub ComboBox1_SelectedIndexChanged(ByVal sender As  
System.Object, ByVal e _  
As System.EventArgs) Handles ComboBox1.SelectedIndexChanged  
End Sub
```

Examples on working with ComboBoxes

Drag a ComboBox and a TextBox control onto the form. We want to display the selection we make in the ComboBox in the TextBox. The code for that looks like this:

```
Private Sub ComboBox1_SelectedIndexChanged(ByVal sender As  
System.Object, ByVal e As _  
System.EventArgs) Handles ComboBox1.SelectedIndexChanged  
TextBox1.Text = ComboBox1.SelectedItem  
'selecting the item from the ComboBox with selected item property  
End Sub
```

Removing items from a ComboBox

You can remove all items or one particular item from the list box part of the ComboBox. Code to remove a particular item by its Index number looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e _  
As System.EventArgs) Handles Button1.Click  
ComboBox1.Items.RemoveAt(4)  
'removing an item by specifying its index  
End Sub
```

Code to remove all items from the combo box.

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
ByVal e As System.EventArgs) Handles Button1.Click  
ComboBox1.Items.Clear()  
'using the clear method to clear the list box  
End Sub
```

Panel

Panels are those controls which contain other controls, say a set of radio buttons. Panels are similar to Groupboxes but the difference is Panels cannot display captions where as GroupBoxes does and also Panels can have scrollbars where as GroupBoxes can't. If the Panels *Enabled* property is set to *False* then the controls which the Panel contains are also disabled. Panels are based on the *ScrollableControl* class.

Coalesce

The properties of the Panel control which are also shared by other controls can be viewed by *clicking here*.

Notable property of the Panel control in the appearance section is the *BorderStyle* property. The default value of the BorderStyle property is set to None. You can select from the predefined list to change a Panels BorderStyle.

Notable property in the layout section is the *AutoScroll* property. Default value is set to False. Set it to True if you want a scrollbar with the Panel.

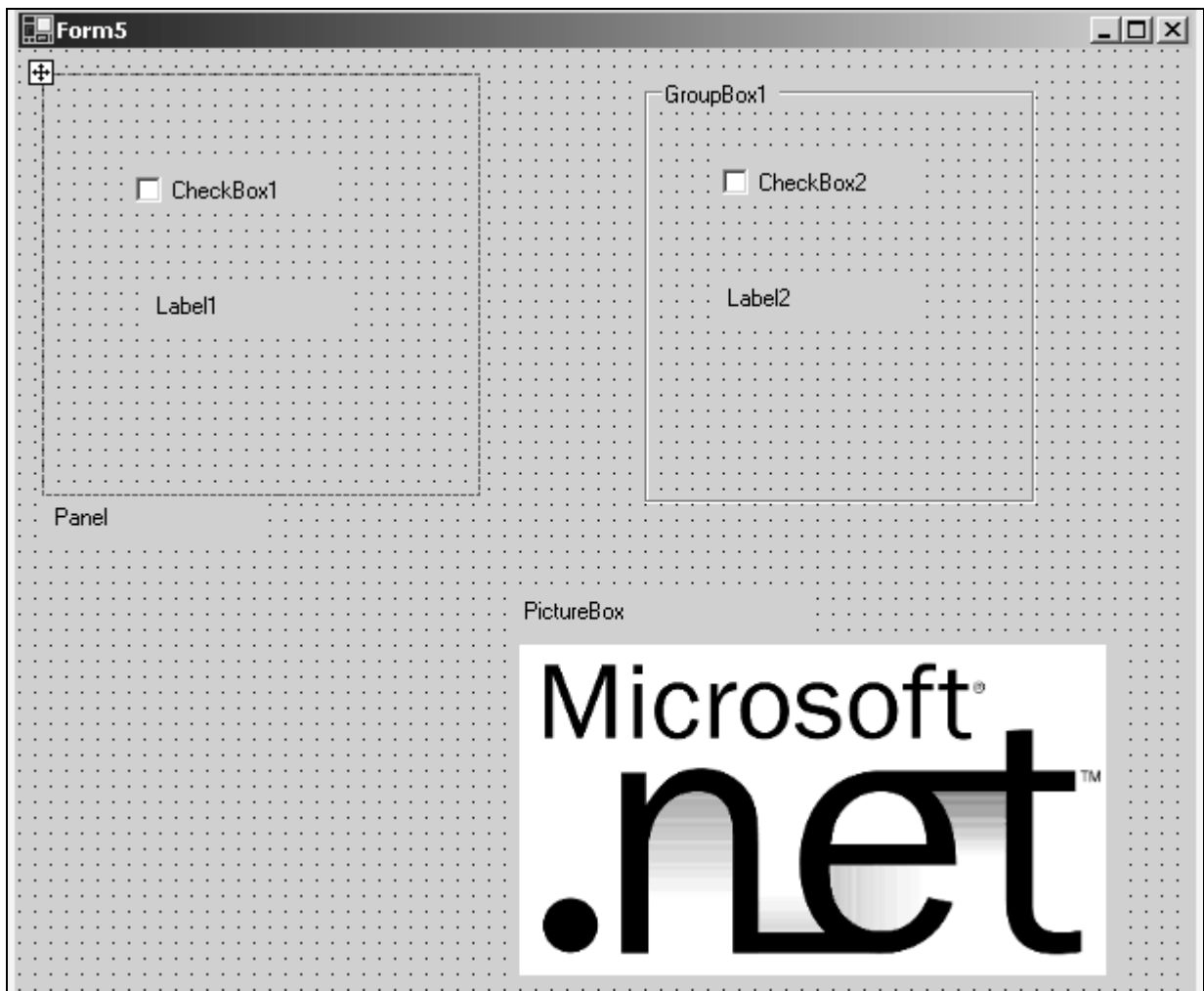
Adding Controls to a Panel

On a form drag a Panel(Panel1) from the toolbox. We want to place some controls, say checkboxes on this Panel. Drag three checkboxes from the toolbox and place them on the Panel. When that is done all the checkboxes in the Panel are together as in a group but can function independently.

Creating a Panel and adding a Label and a Checkbox to it in Code:

```
Private Sub Form3_Load(ByVal sender As System.Object, ByVal e _  
As System.EventArgs) Handles MyBase.Load  
Dim Panel1 As New Panel()  
Dim CheckBox1 As New CheckBox()  
Dim Label1 As New Label()  
Panel1.Location = New Point(30, 60)  
Panel1.Size = New Size(200, 264)  
Panel1.BorderStyle = BorderStyle.Fixed3D  
'setting the borderstyle of the panel  
Me.Controls.Add(Panel1)  
CheckBox1.Size = New Size(95, 45)  
CheckBox1.Location = New Point(20, 30)  
CheckBox1.Text = "Checkbox1"  
Label1.Size = New Size(100, 50)  
Label1.Location = New Point(20, 40)  
Label1.Text = "CheckMe"  
Panel1.Controls.Add(CheckBox1)  
Panel1.Controls.Add(Label1)  
'adding the label and checkbox to the panel  
End Sub
```


Coalesce



GroupBox Control

As said above, Groupboxes can also be used to Group controls. GroupBoxes display a frame around them and also allows us to display captions to it which is not possible with the Panel control. The GroupBox class is based on the *Control* class.

You can view the properties of the GroupBox which are also shared by other controls by *clicking here*.

Creating a GroupBox and adding a Label and a checkBox to it in Code.

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e _  
As System.EventArgs) Handles MyBase.Load  
Dim GroupBox1 As New GroupBox()  
Dim CheckBox1 As New CheckBox()  
Dim Label1 As New Label()  
GroupBox1.Location = New Point(30, 60)  
GroupBox1.Size = New Size(200, 264)  
GroupBox1.Text = "InGroupBox"
```

Coalesce

```
'setting the caption to the groupbox
Me.Controls.Add(GroupBox1)
CheckBox1.Size = New Size(95, 45)
CheckBox1.Location = New Point(20, 30)
CheckBox1.Text = "Checkbox1"
label1.Size = New Size(100, 50)
Label1.Location = New Point(20, 40)
Label1.Text = "CheckMe"
GroupBox1.Controls.Add(CheckBox1)
GroupBox1.Controls.Add(Label1)
'adding the label and checkbox to the groupbox
End Sub
```

PictureBox Control

PictureBoxes are used to display images on them. The images displayed can be anything varying from Bitmap, JPEG, GIF, PNG or any other image format files. The PictureBox control is based on the *Control* class.

The properties of PictureBox which are also shared by other controls can be *viewed here*.

Notable property of the PictureBox Control in the Appearance section of the properties window is the *Image* property which allows to add the image to be displayed on the PictureBox.

Adding Images to PictureBox

Images can be added to the PictureBox with the Image property from the Properties window or by the following code:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click
PictureBox1.Image = Image.FromFile("C:\sample.gif")
'loading the image into the picturebox using the FromFile method of
the image class
'assuming a GIF image named sample in C: drive
End Sub
```

Date TimePicker

Date TimePicker allows us to select date and time. Date TimePicker is based on the *control* class. When we click on the drop-down arrow on this control, it displays a month calendar from which we can make selections. When we make a selection, that selection appears in the textbox part of the Date TimePicker.

Notable Properties of Date TimePicker

The *Format* property in the Appearance section is used to select the format of the date and time selected. Default value is long which displays the date in long format. Other values include short, time and custom

Coalesce

Behavior Section

The *CustomFormat* property allows us to set the format for date and time depending on what we like. To use the CustomFormat property we need to set the Format property to Custom. The *MaxDate* Property allows us to set the maximum date we want the Date TimePicker to hold. Default MaxDate value set by the software is 12/31/9998 . The *MinDate* Property allows us to set the minimum date we want the Date TimePicker to hold. Default MinDate value set by the software is 1/1/1753 .

Month Calendar

The MonthCalendar control allows us to select the date. The difference between a Date TimePicker and Month Calendar is, in MonthCalendar we select the date visually and in Date TimePicker when we want to make a selection we click on the drop-down arrow and select the date from the MonthCalendar which is displayed.

Notable Behavior properties of MonthCalendar

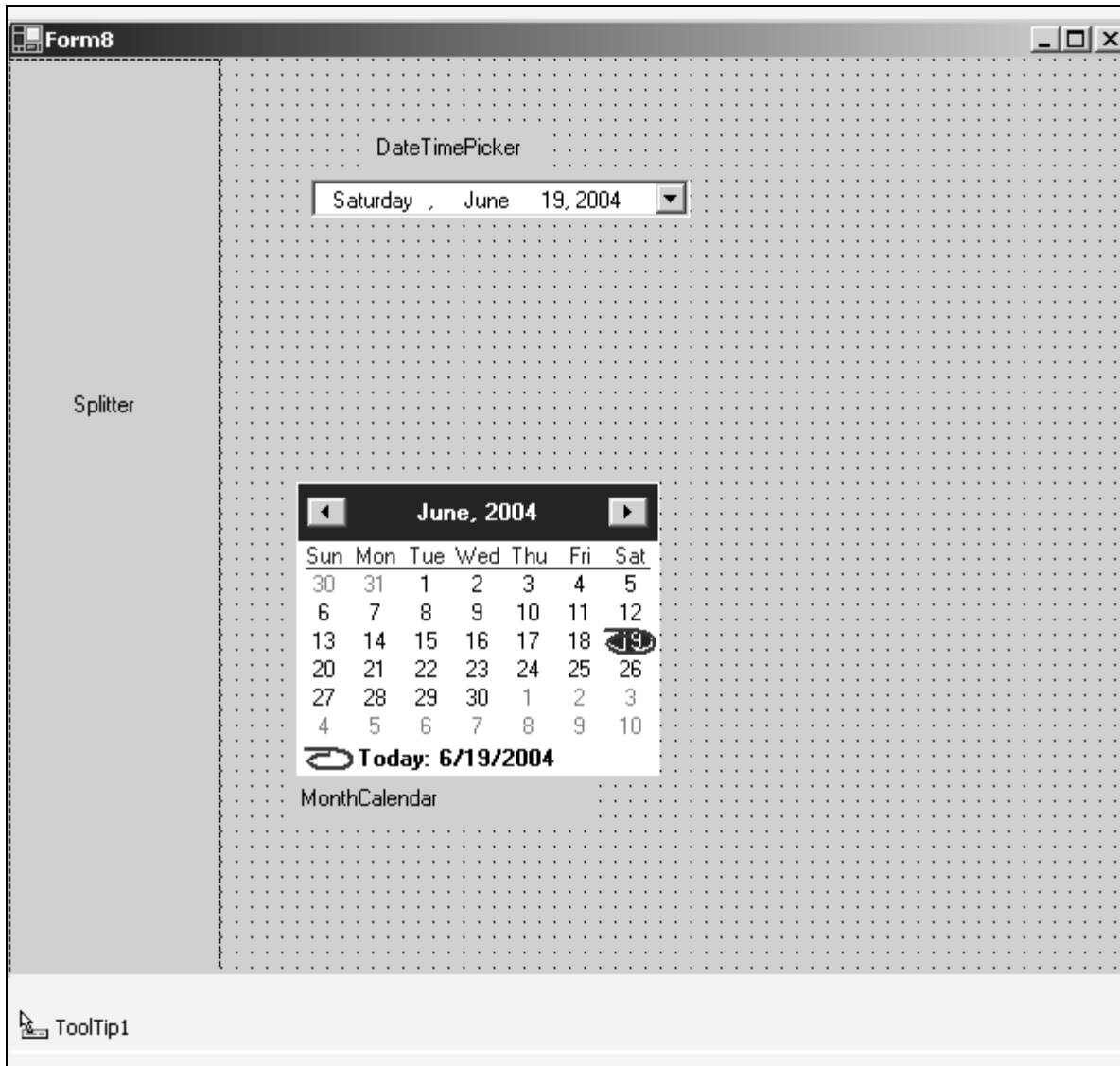
FirstDayOfWeek: Default value is Default which means that the week starts with Sunday as the first day and Saturday as last. You can set the first day of the week depending upon your choice by selecting from the predefined list with this property.

ShowToday: Default value is set to True which displays the current date at the bottom of the Calendar. Setting it to False will hide it.

ShowTodayCircle: Default value is set to True which displays a red circle on the current date. Setting it to False will make the circle disappear.

ShowWeekNumber: Default is False. Setting it to True will display the week number of the current week in the 52 week year. That will be displayed towards the left side of the control.

Coalesce



ToolTip

ToolTips are those small windows which display some text when the mouse is over a control giving a hint about what should be done with that control. ToolTip is not a control but a *component* which means that when we drag a ToolTip from the toolbox onto a form it will not be displayed on the form but will be displayed beneath the form on the component tray, where as controls are displayed on the form. To make ToolTip work with controls we can use its SetToolTip method.

Notable property of the ToolTip is the *Active* property which is set to True by default and which allows the tool tip to be displayed.

Coalesce

Example on Setting a ToolTip

Assume that we have a TextBox on the form and we want to display some text when your mouse is over the TextBox. Say the text that should appear is "Enter Some text". The code for that looks like this:

```
Private Sub Form5_Load(ByVal sender As System.Object, ByVal e _  
As System.EventArgs) Handles MyBase.Load  
    ToolTip1.SetToolTip(TextBox1, "Enter Some text")  
    'using the SetToolTip method to set the tip for the TextBox and the  
    'text that should appear  
End Sub
```

Splitter

The Splitter control is used to resize other controls. The main purpose of Splitter control is to *save space* on the form. Once when we finish working with a particular control we can move it away from its position or resize them with Splitter control. The Splitter control is invisible when we run the program but when the mouse is over it, the mouse cursor changes indicating that it's a Splitter control and it can be resized. This control can be very useful when we are working with controls both at design time and run time (which are not visible at design time). The Splitter control is based on the *Control* class.

Working with Splitter Control.

To work with a Splitter Control we need to make sure that the other control with which this control works is *docked* towards the same side of the container. Let's do that with an example. Assume that we have a TextBox on the form. Drag a Splitter control onto the form. Set the TextBox's dock property to left. If we want to resize the TextBox once we finish using it, set the Splitter's dock property to left (both the controls should be docked towards the same end). When the program is executed and when you pass the mouse over the Splitter control it allows us to resize the TextBox allowing us to move it away from its current position.

Menus

Everyone should be familiar with Menus. Menus (File, Edit, Format etc in all windows applications) are those that allow us to make a selection when we want to perform some action with the application, for example, to format the text, open a new file, print and so on. In VB.NET *MainMenu* is the container for the Menu structure of the form. Menus are made of *MenuItem* objects that represent individual parts of a menu (like File->New, Open, Save, Save As etc). The two main classes involved in menu handling are, MainMenu and MenuItem. The MainMenu class let's us assign objects to a form's menu class and MenuItem is the class which supports the items in a menu system. Menus like File, Edit, Format etc and the items in those Menus are supported by this MenuItem class. It's this Menu Item's click event that makes these Menus work. For a MenuItem to be displayed, we need to add it to a MainMenu object.

Event of the MenuItem

The default event of the MenuItem is the menu_click event which looks like this in code:

Coalesce

```
Private Sub MenuItem1_Click(ByVal sender As System.Object,  
ByVal e As System.EventArgs) Handles MenuItem1.Click  
  
End Sub
```

Notable properties of the MenuItem class are summarized below.

Under the Miscellaneous Section of the properties window:

Checked: Default value is set to False. Changing it to True makes a checkmark appear towards the left of the Menu.

DefaultItem: Default value is set to False. Changing it to True makes this menu item default menu item.

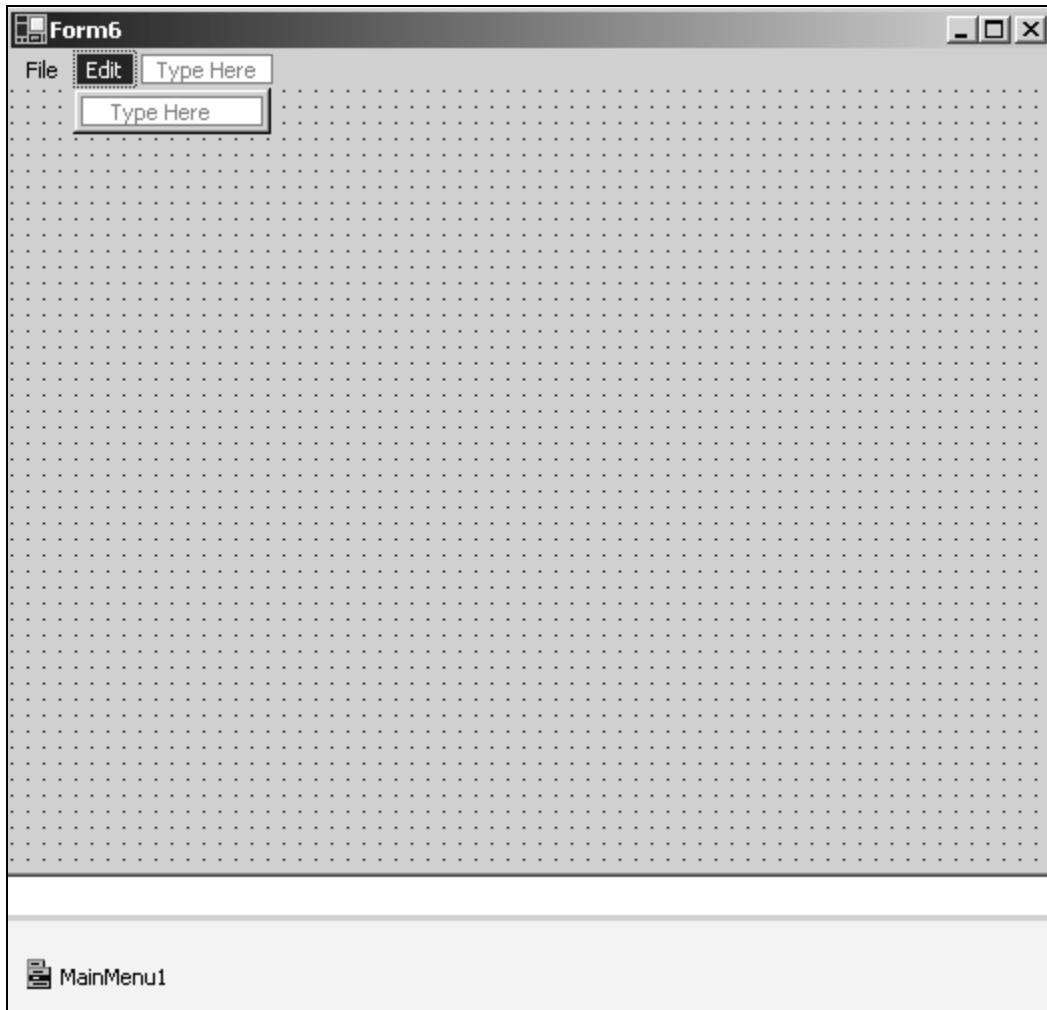
RadioCheck: Changing it to True makes a menu item display a radio button instead of a checkmark.

Shortcut: Enables to set a short cut key from a list of available shortcuts for the menu item.

Working with Menus

Creating Menus is simple, just drag a MainMenu component from the toolbar onto the form. When you add a MainMenu component to the form it appears in the component tray below the form. Windows form designer will add the MenuItem's for this by default, you need not add this. Once when you finish adding a MainMenu component to the form you will notice something called "*TypeHere*" towards the top left corner of the form. To create a menu all you have to do is click on the "TypeHere" text which opens up a small textbox allowing you to enter text for the menu. You can view that in the image below. You can use the arrow keys on the keyboard to create a submenu or add other items to that menu or click on the first menu item and use the left/right arrow keys on the keyboard to create a new menu item. That's all it takes to add a menu to the form.

Coalesce



Working with an example

Let's work with an example to understand Menus. Drag a MainMenu and a TextBox onto the form. In the "Type Here" part, type File and under file type "New" and "Exit". Our intention here is to display "Welcome to Menus" in the TextBox when "New" is clicked and close the form when "Exit" is clicked. The Menu which we will create should look like this File->New, Exit (New and Exit under File). The code for that looks like this:

```
Public Class Form3 Inherits System.Windows.Forms.Form
#Region " Windows Form Designer generated code "
Private Sub MenuItem2_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles MenuItem2.Click
    TextBox1.Text = "Welcome to Menus"
End Sub

Private Sub MenuItem3_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles MenuItem3.Click
    Me.Close()
'Me refers to the current object which is the form
End Sub
```

Coalesce

End Class

'By default, File is assigned MenuItem1, New as MenuItem2 and Exit as MenuItem3

Menus and Dialog Boxes are often used together in combination.

Dialog Boxes

Visual Basic.NET comes with many built-in dialog boxes which allows us to create our own FileOpen, FileSave, Print, PrintPreview style dialogs, much like what we see in all other windows applications. To make a dialog box visible at run time we use the dialog box's ShowDialog method. The Dialog Boxes which come with Visual Basic.NET are : OpenFileDialog, SaveFileDialog, FontDialog, ColorDialog, PrintDialog, PrintPreviewDialog and PageSetupDialog. We will be working with OpenFile, SaveFile, PrintDialog and PrintPreviewDialog's in this section. The return values of all these dialog boxes which will determine which button a user clicks are: Abort, Cancel, Ignore, No, None, OK, Return, Retry and Yes.

OpenFileDialog

OpenFileDialog's are supported by the *OpenFileDialog* class and they allow us to select a file to be opened. Notable properties of the OpenFileDialog as visible under the Misc Section in the properties window are the as follows:

InitialDirectory: This property allows to set the directory to be opened when we select the OpenFileDialog.

MultiSelect: This property when set to 'True allows to select multiple file extensions.

RestoreDirectory: If 'True, this property restores the original directory before closing.

Title: This property allows to set a title for the file dialog box.

SaveFileDialog

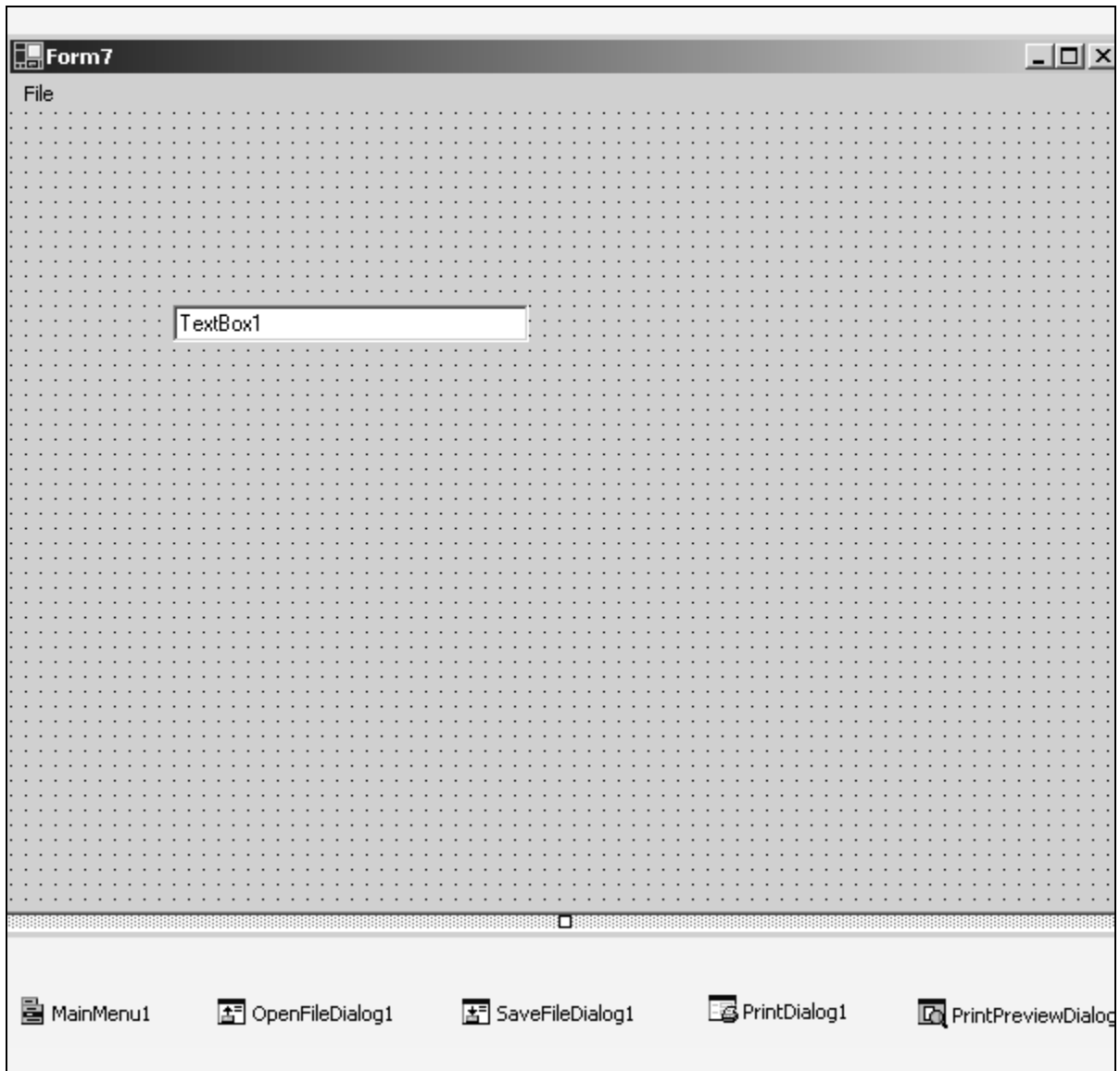
SaveFileDialog's are supported by the *SaveFileDialog* class and they allow us to save the file in the specified location. Notable properties of the SaveFileDialog as visible in the Misc Section are:

FileName: This property allows to set the file name to which it will be saved as.

OverwritePromopt: This property displays a warning if we choose to save to a name that already exists.

ValidateNames: This property is used to specify whether the dialog box accepts only valid file names.

Coalesce



Print Dialog

PrintDialog's are supported by the *PrintDialog* class and they allow us to print the document. To print a document we need to set the document property of the PrintDialog to PrintDocument object and PrinterSettings to PrinterSettings object, which also should be dragged from the toolbox when working with PrintDialog. The PrinterSettings property allows us to specify the number of copies to be printed, the printer to be used, etc. To print the document we use the PrintDocument object's Print method. Notable properties of the PrintDialog as visible in the Misc section of the properties window are as follows:

AllowSelection: If set to True, enables a selection radio button.

AllowSomePages: If set to True, enables a From...To... radio button.

Document: Used to set the PrintDocument.

Coalesce

PrintPreviewDialog

PrintPreviewDialog's are supported by the *PrintPreviewDialog* class and allow us to preview the document before printing, much like the print preview feature in windows applications. PrintPreview's main property is the Document property that sets the document to be previewed.

Putting Dialog Boxes to Work

We will work with OpenFileDialog, SaveFile, Print and Print Preview Dialog's in this section. From the toolbox drag a MainMenu, TextBox, RichTextBox, OpenFileDialog, SaveFileDialog, PrintDialog, PrintDocument and a PrintPreviewDialog onto the form. Our intention here is to display the path of the file in the TextBox which we open with the OpenFileDialog, path of the file which we choose to save using the SaveFileDialog in the RichTextBox and print and preview the document. In the "Type Here" part of the MainMenu, type File and using the down arrow keys on the keyboard start typing Open, Save, Print, PrintPreview under the File menu. It should look like this: File-> Open, Save, Print, PrintPreview. We will assign OpenFileDialog to Open, SaveFileDialog to Save, PrintDialog to Print and PrintPreviewDialog to Preview under File Menu. The code to get the desired functionality looks like this:

```
Public Class Form1 Inherits System.Windows.Forms.Form
'Windows Form Designer generated code
Private Sub MenuItem2_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs)_
Handles MenuItem2.Click
If OpenFileDialog1.ShowDialog() <> DialogResult.Cancel Then
TextBox1.Text = OpenFileDialog1.FileName
End If
End Sub

Private Sub MenuItem3_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs)_
Handles MenuItem3.Click
If SaveFileDialog1.ShowDialog <> DialogResult.Cancel Then
RichTextBox1.Text = "You selected to save in the location" &
SaveFileDialog1.FileName
End If
End Sub

Private Sub MenuItem4_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs)_
Handles MenuItem4.Click
PrintDialog1.Document = PrintDocument1
Try
PrintDialog1.PrinterSettings = PrintDocument1.PrinterSettings
If PrintDialog1.ShowDialog = DialogResult.OK Then
PrintDocument1.PrinterSettings = PrintDialog1.PrinterSettings
PrintDocument1.Print()
End If
Catch
End Try
```

Coalesce

```
End Sub

Private Sub MenuItem5_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs)_
Handles MenuItem5.Click
PrintPreviewDialog1.Document = PrintDocument1
PrintPreviewDialog1.ShowDialog()
End Sub
End Class
```

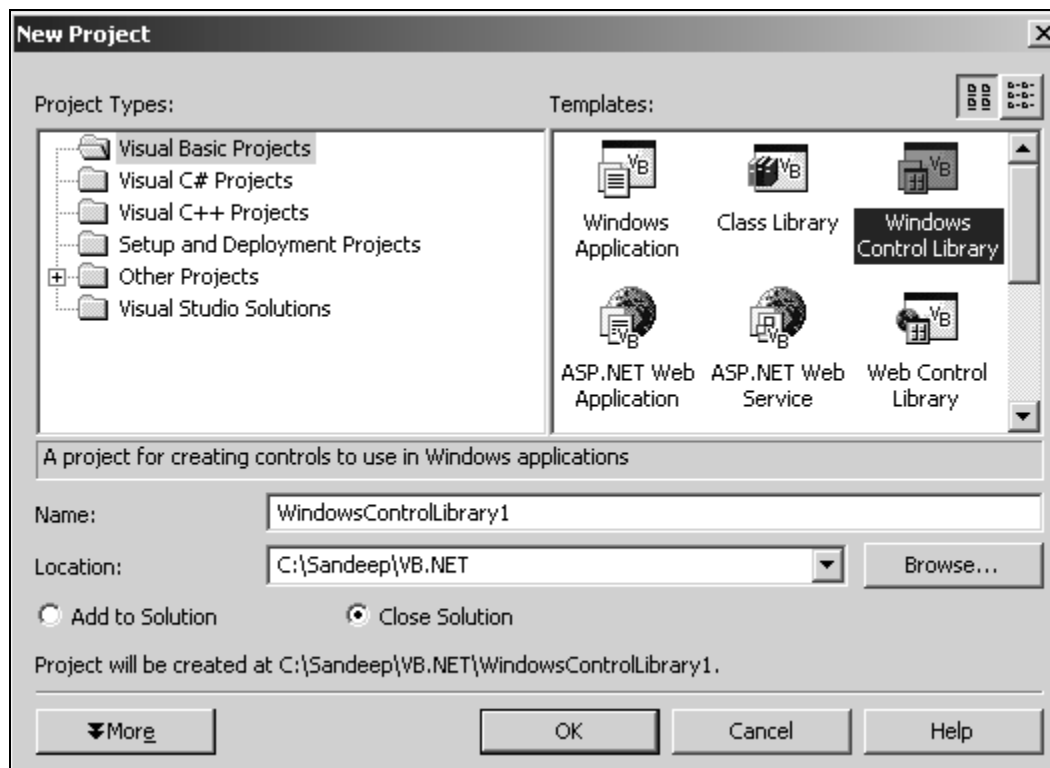
When you run the code, the path of the file you open from the File menu will be displayed in the Textbox and the path of the file you choose to save the file will be displayed in the RichTextBox.

Creating User Controls

User Controls are the controls which are created by the user and are based on the class *System.Windows.Forms.UserControl*. Like standard controls, user controls support properties, methods and events. Once a user control is created it can be added to any form or any number of forms like all other controls.

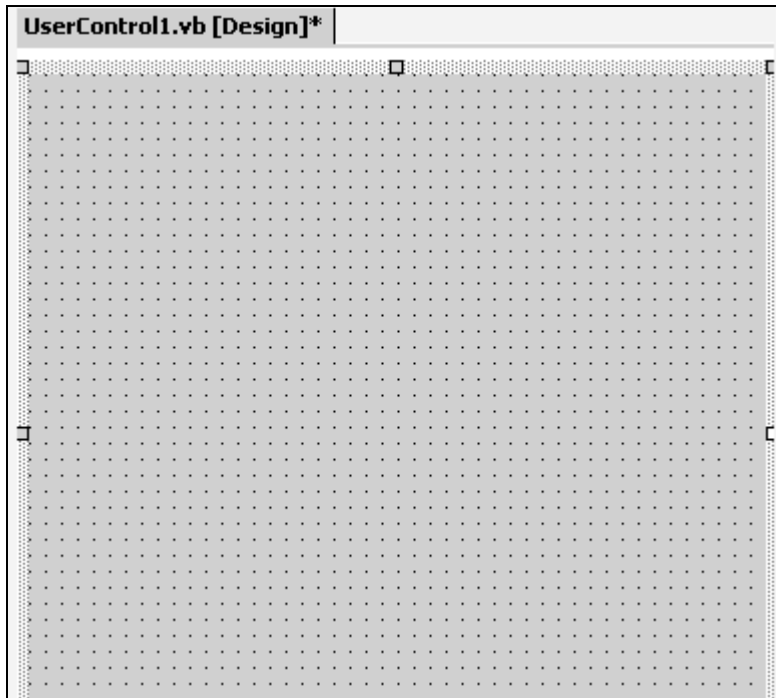
Creating a User Control

To create a user control select *File->New->Project->Visual Basic Projects* and select *Windows Control Library* from the templates and click OK. Alternatively, you can add user control to the existing project by selecting *Project->Add User Control* menu. The image below shows how to add a User Control project.



Coalesce

The form that opens after clicking OK looks like the image below.



Example For Creating a User Control

Drag a Label and a TextBox control from the toolbox onto the new user control form. The image below displays that.



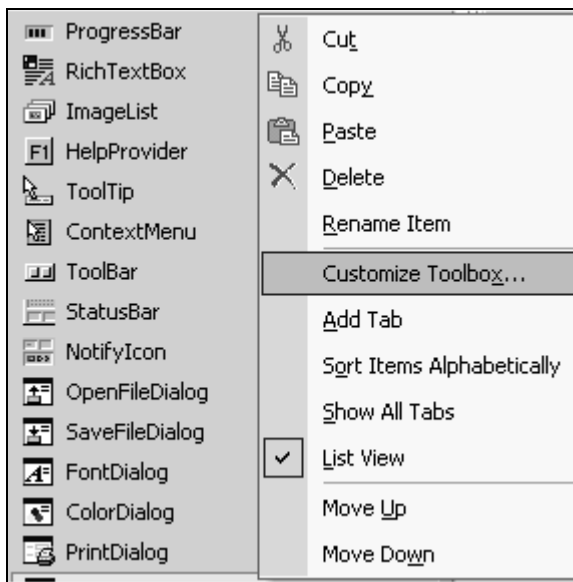
Double-click the user control form to open its code behind file. In the code behind file type the following code below the Load event of UserControl1 to set the property of the user control which is being created.

```
Public Property sanText() As String
```

Coalesce

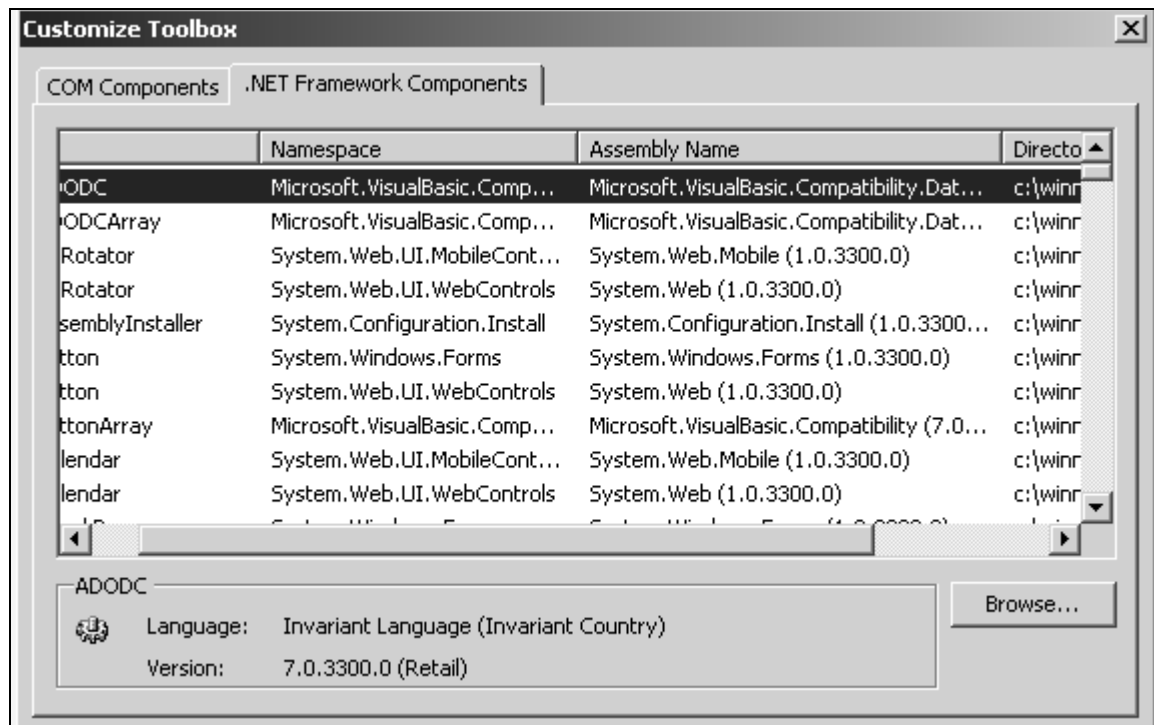
```
Get
sanText = TextBox1.Text
End Get
Set(ByVal Value As String)
TextBox1.Text = Value
End Set
End Property
Public Property sanLbl() As String
Get
sanLbl = Label1.Text
End Get
Set(ByVal Value As String)
Label1.Text = Value
End Set
End Property
```

The above code implements the `sanText()` and `sanLbl()` properties with a property get/set pair. Once typing the code is done we need to build the solution using *Build->Build Solution* from the main menu to create the *.dll (Dynamic Link Library)* file to which we refer to. The dll file also makes this user control available to other projects. Having finished building the solution we next need to add this user control to the toolbox to make it available to other projects and forms. To do that open a windows form or use an existing form. Right-click on the *Windows Form* tab section on the toolbox and click on *Customize Toolbox* as shown in the image below.



Clicking on *Customize Toolbox* opens "Customize Toolbox" dialogue box as shown in the image below.

Coalesce



On this window select the .NET Framework components tab which displays a list. Click browse to add the dll of the user control we created and select the dll file from the bin directory of the Windows Control library project. Once that is done click OK. The dll of the user control gets added to the .NET Framework Components. Select that and click OK. The user control which we created gets added to the toolbox. Now this control can be used like other controls in the toolbox.

Alternatively, the user control can be added to the toolbox in another way. To do that, right-click on the References item in the Solution Explorer window and select Add Reference which opens the Add Reference dialogue box. Select the projects tab and double-click the User Control item. That adds item to the selected components box at the bottom of the dialog. click OK to finish.

Using the User Control

The user control, UserControl11 which we created can be added to any form like all other controls. Open a new form and drag this user control from toolbox to the form. Drag a Button from the toolbox and place that on the form. Double-click on the Button to open it's Click event. Place the following code in the Click event of the Button:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e  
As System.EventArgs) Handles Button1.Click  
    MessageBox.Show("TextBox" & UserControl11.sanText())  
    MessageBox.Show("Label1" & UserControl11.sanlb())  
End Sub
```

Run the code and enter some text in the TextBox. When you click the Button, the text entered in the TextBox is displayed in a MessageBox. That's the way User Controls are used after they are created.

Coalesce

Chapter 15

I/O Handling

File Handling

File handling in Visual Basic is based on *System.IO* namespace with a class library that supports string, character and file manipulation. These classes contain properties, methods and events for creating, copying, moving, and deleting files. Since both strings and numeric data types are supported, they also allow us to incorporate data types in files. The most commonly used classes are *FileStream*, *BinaryReader*, *BinaryWriter*, *StreamReader* and *StreamWriter*.

FileStream Class

This class provides access to standard input and output files. We use the members of *FileAccess*, *FileMode* and *FileShare* Enumerations with the constructors of this class to create or open a file. After a file is opened its *FileStream* object can be passed to the *Binary Reader*, *BinaryWriter*, *Streamreader* and *StreamWriter* classes to work with the data in the file. We can also use the *FileStreamSeek* method to move to various locations in a file which allows to break a file into records each of the same length.

StreamReader and StreamWriter Class

The *StreamReader* and *StreamWriter* classes enables us to read or write a sequential stream of characters to or from a file.

BinaryReader and BinaryWriter Class

The *BinaryReader* and *BinaryWriter* classes reads and writes binary data that is in the raw 0's and 1's, which is how data is stored on the computer.

The following examples puts some code to work with textual data using *FileStream* and *StreamReader* and *StreamWriter* classes.

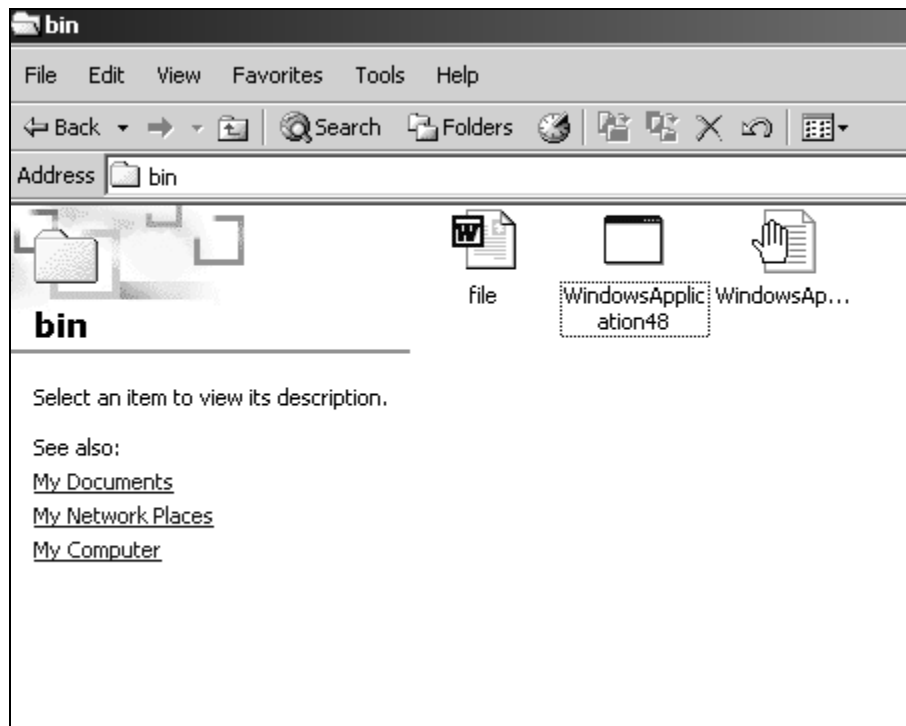
Code to create a File

```
Imports System.IO
'Namespace required to be imported to work with files
Public Class Form1 Inherits System.Windows.Forms.Form
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e_
```

Coalesce

```
As System.EventArgs) Handles MyBase.Load
Dim fs as New FileStream("file.doc", FileMode.Create,
FileAccess.Write)
'declaring a FileStream and creating a word document file named file
with access mode of writing
Dim s as new StreamWriter(fs)
'creating a new StreamWriter and passing the filestream object fs as
argument
s.BaseStream.Seek(0,SeekOrigin.End)
'the seek method is used to move the cursor to next position to avoid
text to be overwritten
s.WriteLine("This is an example of using file handling concepts in VB
.NET.")
s.WriteLine("This concept is interesting.")
'writing text to the newly created file
s.Close()
'closing the file
End Sub
End Class
```

The default location where the files we create are saved is the bin directory of the Windows Application with which we are working. The image below displays that.



Code to create a file and read from it

```
Imports System.IO
'Namespace required to be imported to work with files
Public Class Form1 Inherits System.Windows.Forms.Form
Private Sub Button1_Click(ByVal....., Byval.....)Handles Button1.Click
```

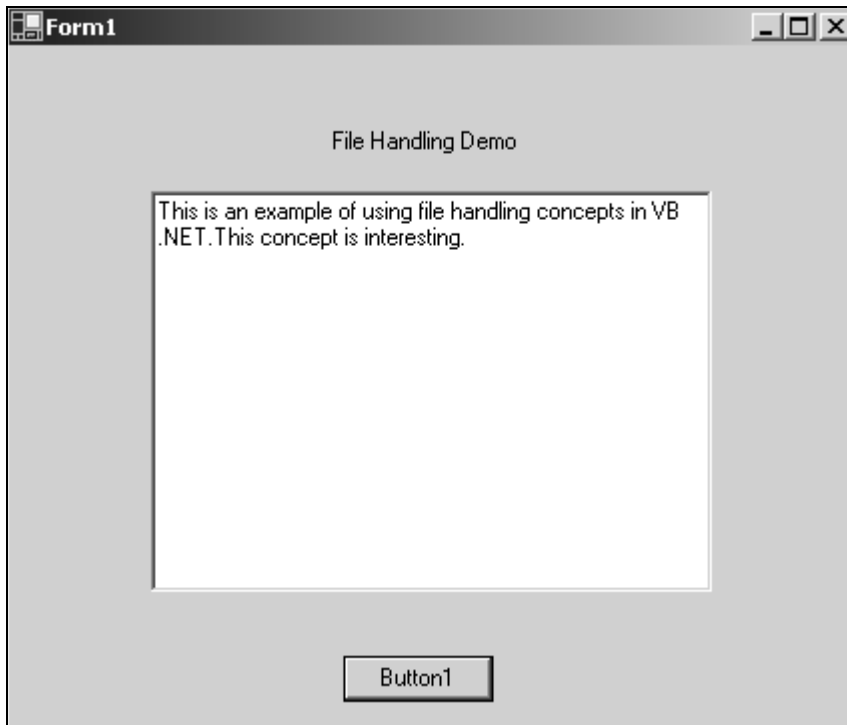

Coalesce

```
Dim fs as New FileStream("file.doc", FileMode.Create,
FileAccess.Write)
'declaring a FileStream and creating a document file named file with
access mode of writing
Dim s as new StreamWriter(fs)
'creating a new StreamWriter and passing the filestream object fs as
argument
s.WriteLine("This is an example of using file handling concepts in VB
.NET.")
s.WriteLine("This concept is interesting.")
'writing text to the newly created file
s.Close()
'closing the file

fs=New FileStream("file.doc",FileMode.Open,FileAccess.Read)
'declaring a FileStream to open the file named file.doc with access
mode of reading
Dim d as new StreamReader(fs)
'creating a new StreamReader and passing the filestream object fs as
argument
d.BaseStream.Seek(0,SeekOrigin.Begin)
'Seek method is used to move the cursor to different positions in a file,
in this code, to the beginning
while d.peek()>-1
'peek method of StreamReader object tells how much more data is left
in the file
RichTextbox1.Text &= d.readLine()
'displaying text from doc file in the RichTextBox
End while
d.close()
End Sub
```

The image below displays output of the above code .

Coalesce



Working with the File and Directory Class

We will work with the *File* and *Directory* classes in this section. We will create a directory and copy a file into that newly created directory.

File Class

The File class in VB.NET allows us to work with files allowing to copy, delete and create files.

Directory Class

The Directory class in VB.NET allows us to create and work with Folders and Directories. With this class we can create, edit and delete folders and also maintain drives on the machine.

Code to work with File and Directory Class

What we will be doing in this example code is, we will create a directory and then copy a file into that new directory. To create a new directory we should use the Directory class's *CreateDirectory* method. Now drag two Button's (Button1, Button2), a TextBox and a OpenFileDialog control from the toolbar onto the Form. We will use the TextBox to specify a location to create the Directory when Button1 is clicked and the OpenFileDialog control to open a file and copy it into the newly created Directory when Button2 is clicked. The namespace to be imported is *System.IO*. The code for that looks like this:

```
Imports System.IO
```

Coalesce

```
Public Class Form1 Inherits System.Windows.Forms.Form

    'Windows Form Designer Generated Code

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles MyBase.Load

    End Sub

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles Button1.Click

    Try
        Directory.CreateDirectory(TextBox1.Text)
        'Creating a directory by specifying a path in the TextBox, say you specified
c:\examples
        'Instead of using a TextBox you can directly type the location of the directory like
this
        'Directory.CreateDirectory("c:\examples")

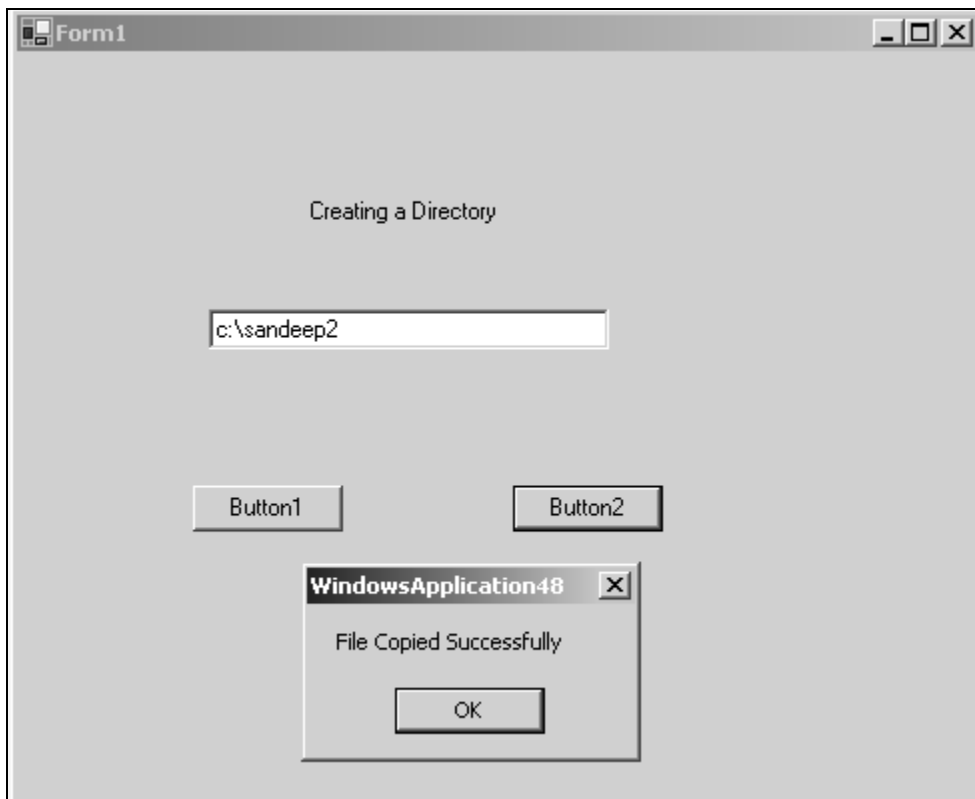
    Catch
    End Try
    MsgBox("Done")
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles Button2.Click

    Try
        If OpenFileDialog1.ShowDialog <> DialogResult.Cancel Then
            File.Copy(OpenFileDialog1.FileName, TextBox1.Text & "\" & _
OpenFileDialog1.FileName.Substring(OpenFileDialog1.FileName.LastIndexOf("\")))
            'The above line of code uses OpenFileDialog control to open a dialog box where you
'can select a file to copy into the newly created directory
        End If
    Catch
    End Try
    MsgBox("File Copied Successfully")
    End Sub
End Class
```

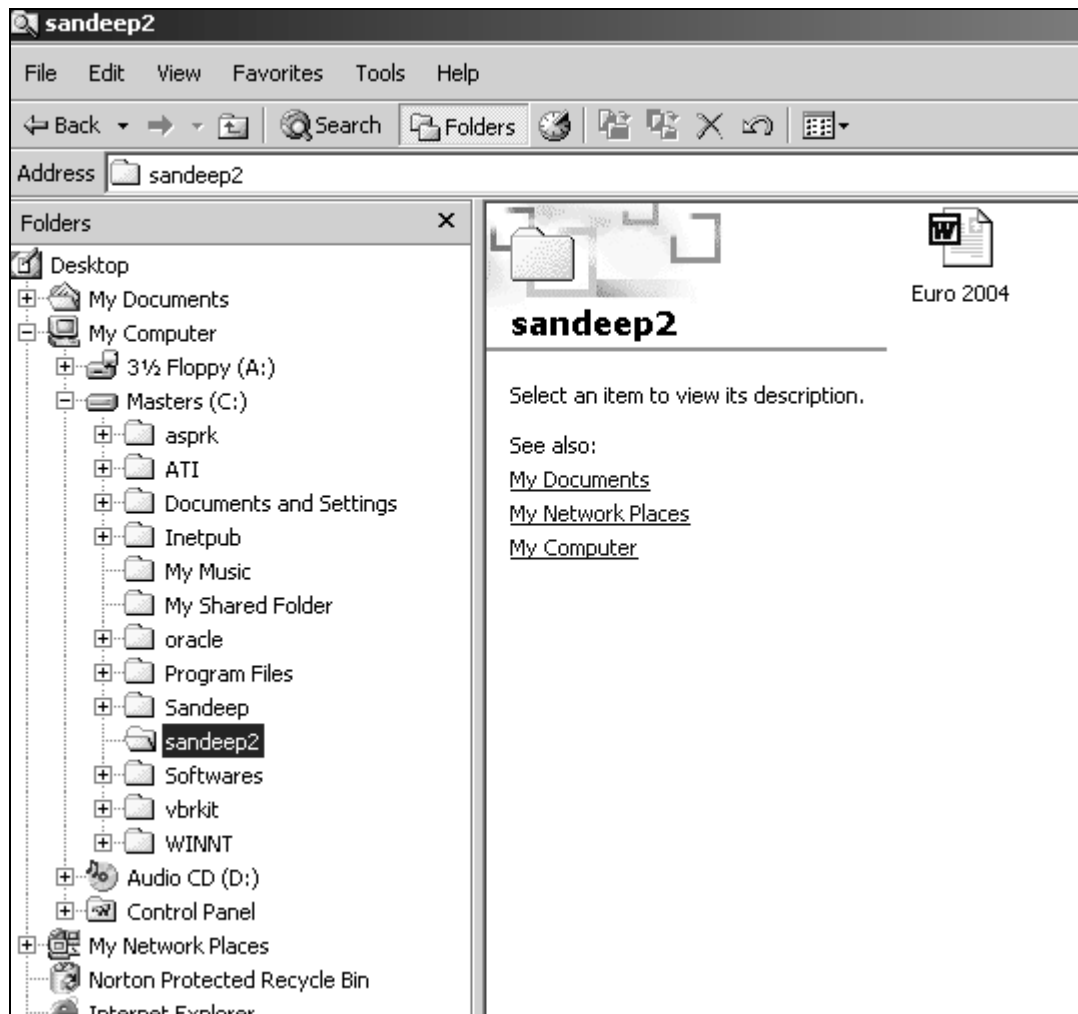
That's all it takes to create a directory and copy a file into the newly created directory in VB.NET. The image below shows output from above code.

Coalesce



The image below shows the directory I created and the file I copied to the new directory using the code above.

Coalesce



Multithreading

Multithreading gives programs the ability to do several things at a time. Each stream of execution is called a thread. Multithreads are used to divide lengthy tasks that would otherwise abort programs. Threads are mainly used to utilize the processor to a maximum extent by avoiding its idle time. Threading lets a program seem as if it is executing several tasks at once. What actually happens is, the time gets divided by the computer into parts and when a new thread starts, that thread gets a portion of the divided time. Threads in VB.NET are based on the namespace, *System.Threading*

Creating Threads

To create thread, let's go with an example. The following example is an extract from Steven Holzner's book *Programming with Visual Basic.NET- Black Book*. Open a new windows application and name it as Thread and add a class named count1 using the *Projects->Add Class item*. This class will count from 1 to a specified value in a data member named CountTo, when you call the Count method. After the count has reached the value in CountTo, a FinishedCounting event will occur. The code for the Count class looks like this:

Coalesce

```
Public Class Count1
Public CountTo as Integer
Public event FinishedCounting(By Val NumberOfMatches as Integer)
Sub Count()
Dim ind,tot as Integer
tot=0
For ind=1 to CountTo
tot+=1
Next ind
RaiseEvent FinishedCounting(tot)
'makes the FinishedCounting event to occur
End Sub
End Class
```

To make use of this class with a new thread, go back to the main form and create an object of this class, counter1 and a new thread Thread1. The code looks like this:

```
Public Class Form1 Inherits System.Windows.Forms.Form
Dim counter1 as new Count1()
Dim Thread1 as New System.Threading.Thread(Address of
counter.Count)
```

Place a Button and two TextBoxes(TextBox1,TextBox2) on the form. Enter a number in TextBox1 and the reason for entering is, when you click the Button, the code will read the value specified in TextBox1 and displays that in TextBox2, with threading. The code for that looks like this:

```
Public Class Form1 Inherits System.Windows.Forms.Form
Dim counter1 as new Count1()
Dim Thread1 as New System.Threading.Thread(Address of
counter.Count)
Private Sub Button1_Click(ByVal sender as System.Object, ByVal e as
System.EventArgs)_
Handles Button1.Click
TextBox2.Text=" "
counter1.CountTo=TextBox1.Text
AddHandler
counter1.FinishedCounting,AddressOfFinishedCountingEventHandler
'adding handler to handle FinishedCounting Event
Thread1.Start()
'starting the thread
End Sub
Sub FinishedCountingEventHandler(ByVal Count as Integer)
'FinishedCountingEventHandler
TextBox2.Text=Count
End Sub
```

The result of the above code displays the value entered in TextBox1, in TextBox2, the difference being the Thread counting the value from 1 to the value entered in TextBox1.

Coalesce

Suspending a Thread



Threads can be suspended. Suspending a thread stops it temporarily. Working with the example in the previous section, add a new button Button2 to the main form. When this button is clicked the thread is suspended. The code for that looks like this:

```
Private Sub Button2_Click(ByVal sender as System.Object, ByVal e as  
System.EventArgs)_  
Handles Button2.Click  
Thread1.Suspend()  
End Sub
```

Resuming a Thread

Threads can be resumed after they are suspended. With the example above, add a new button Button3 to the main form. When this button is clicked the thread is resumed from suspension. The code for that looks like this:

```
Private Sub Button3_Click(ByVal sender as System.Object, ByVal e as  
System.EventArgs)_  
Handles Button3.Click  
Thread1.Resume()  
End Sub
```

Making a Thread Sleep

Threads can be made to sleep which means that they can be suspended over a specific period of time. Sleeping a thread is achieved by passing the time (in milliseconds, 1/1000 of a second) to the thread's sleep method. With the example above, add a new button Button4 to the main form. When this button is clicked the thread is stopped. The code for that looks like this:

```
Private Sub Button4_Click(ByVal sender as System.Object, ByVal e as  
System.EventArgs)_  
Handles Button4.Click  
Thread1.Sleep(100/1000)  
End Sub
```

Stopping a Thread

Threads can be stopped with it's abort method. With the example above, add a new button Button5 to the main form. When this button is clicked the thread is stopped. The code for that looks like this:

```
Private Sub Button5_Click(ByVal sender as System.Object, ByVal e as  
System.EventArgs)_  
Handles Button5.Click  
Thread1.Abort()  
End Sub
```

Coalesce

Thread Priorities

Threads can also be assigned priority for execution. Thread priority can be set by the thread's Priority property and assigning a value from predefined Thread Priority enumeration.

Values for Thread Priority:

- ▶ Above Normal -> Gives thread higher priority
 - ▶ Below Normal -> Gives thread lower priority
 - ▶ Normal -> Gives thread normal priority
 - ▶ Lowest -> Gives thread lowest priority
 - ▶ Highest -> Gives thread highest priority
- ▶ Highest -> Gives thread highest priority Working with the above example, add a new button Button6 to the main form. When this button is clicked the thread is assigned Highest priority .The code for that looks like this:

```
Private Sub Button6_Click(ByVal sender as System.Object, ByVal e as  
System.EventArgs)_  
Handles Button6.Click  
Thread1.Priority=System.Threading.ThreadPriority.Highest  
'setting Highest priority for the thread  
End Sub
```


Chapter 16

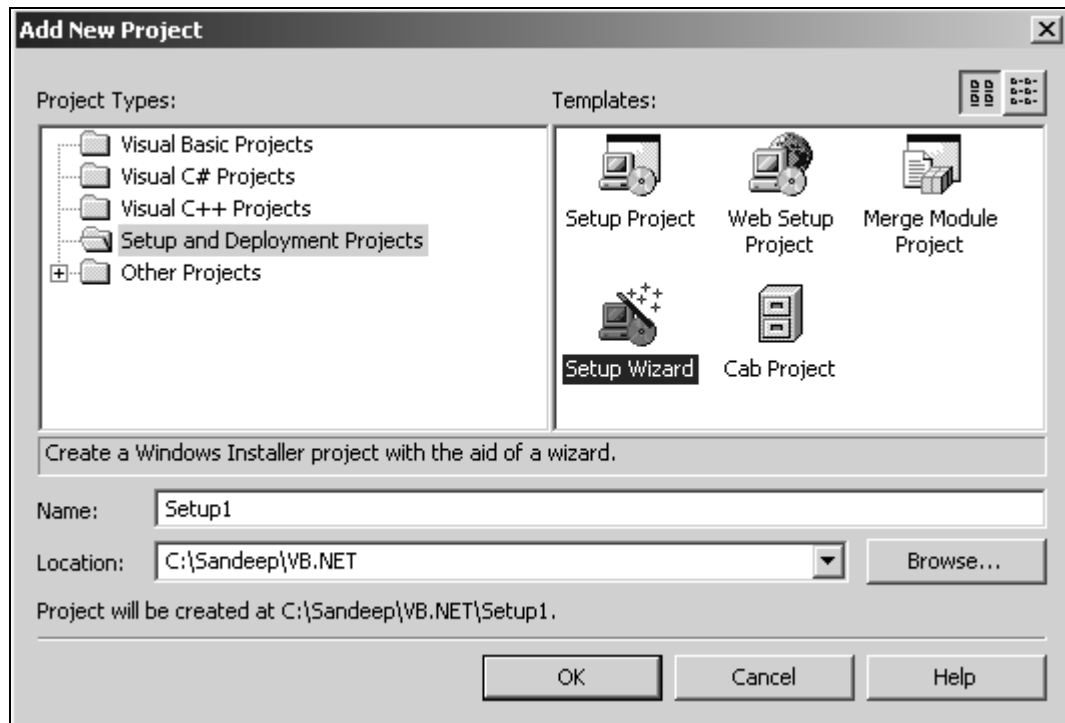
Deploying Applications

Once an application is developed and if we want to distribute that application, we need to deploy that. Deployment is the process where we create an executable file which can be installed on any machine where the application can run. We can use the built-in deployment feature which comes with Visual Basic to create a Windows Installer file - a *.msi* file for the purpose of deploying applications.

Let's look at the process with an example. Let's assume we have a form with a TextBox and a Button. When the Button is clicked the TextBox should display "This application is Deployed". Let's name this application as Deploy. The code for the click event of the Button looks like this:

```
Private Sub Button1_Click(By Val sender as System.Object, By Val e
as System.EventArgs) _
Handles Button1.Click
    TextBox1.Text="This application is Deployed"
End Sub
```

Next, we need to create an executable file for this application. To do that select *Build->Build* from the main menu which builds Deploy.exe. Next, we need to create an installer file for Deploy (which is the example) which is a file with *.msi* extension. To do that, select *File->Add Project->New Project* which opens the new project dialogue. Select "*Setup and Deployment Projects*" icon in the projects type box and the *Setup Wizard* in the templates box. It looks like the image below.



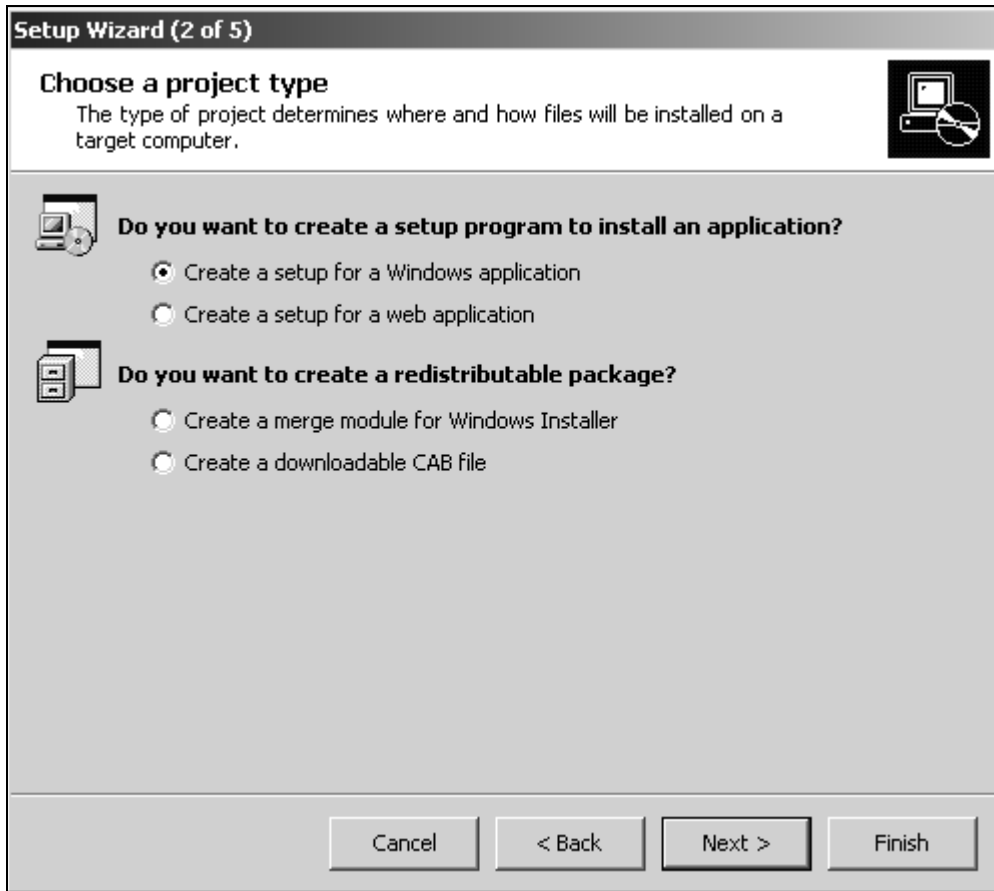
Coalesce

Click OK to open the Setup Wizard. The Setup wizard window looks like the image below.



Click next on the above pane to take you to second pane in the Wizard. The new pane allows us to create deployment projects both for Windows and Web Applications. Here, select the radio button which says *"Create a setup for Windows Application"* as this is deploying a windows Application and click next. It looks like the image below.

Coalesce



Clicking next opens a new pane which has options like Deploying only primary output from the project or both the project and source code or content files. Check the checkbox which you want, in this case check the checkbox that says *"Primary Output from Deploy"* and click next. It looks like the image below.

Coalesce

Setup Wizard (3 of 5)

Choose project outputs to include
You can include outputs from other projects in your solution.

Which project output groups do you want to include?

- ☒ Primary output from Deploy
- ☐ Localized resources from Deploy
- ☐ Debug Symbols from Deploy
- ☐ Content Files from Deploy
- ☐ Source Files from Deploy

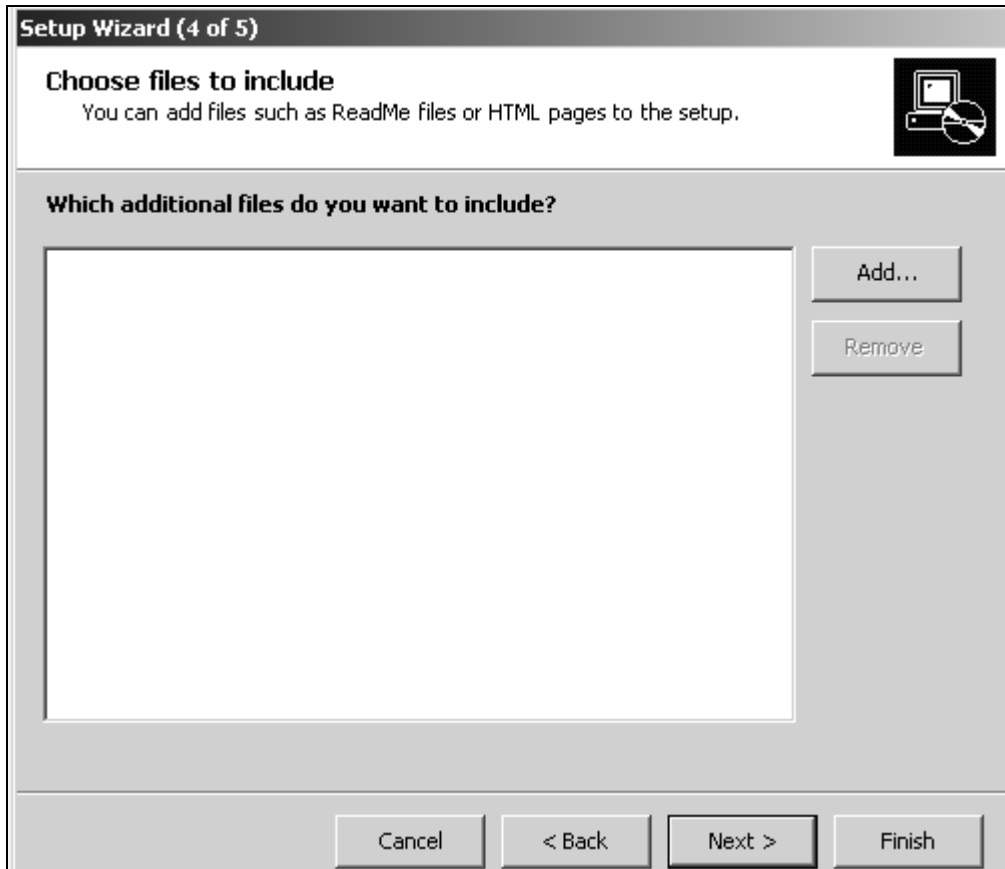
Description:

Contains the DLL or EXE built by the project.

Cancel < Back Next > Finish

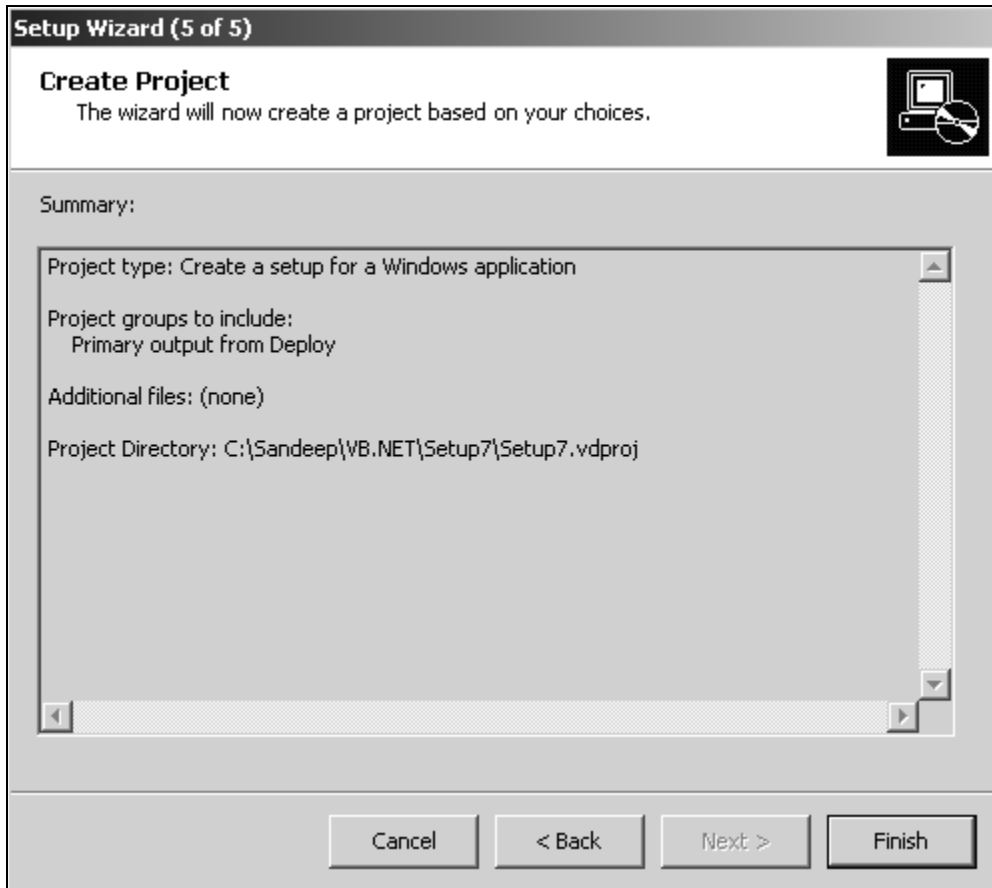
Clicking next opens a new pane which asks if you want any additional files to be added. If you wish, you can include other files, like an icon for the application. In this example don't include any files and click next. It looks like the image below.

Coalesce



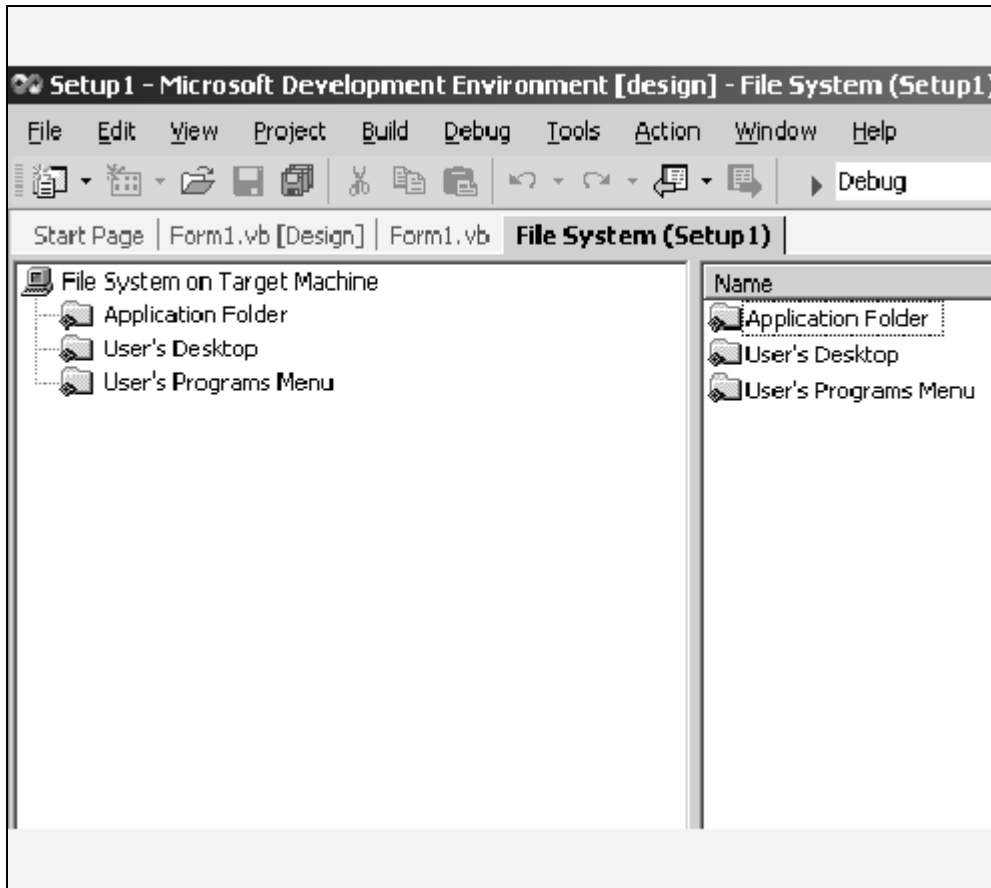
Doing that brings up the last pane of the Setup Wizard which looks like the image below. Click Finish on this pane.

Coalesce



Clicking finish opens up a File System window which looks like the image below.

Coalesce

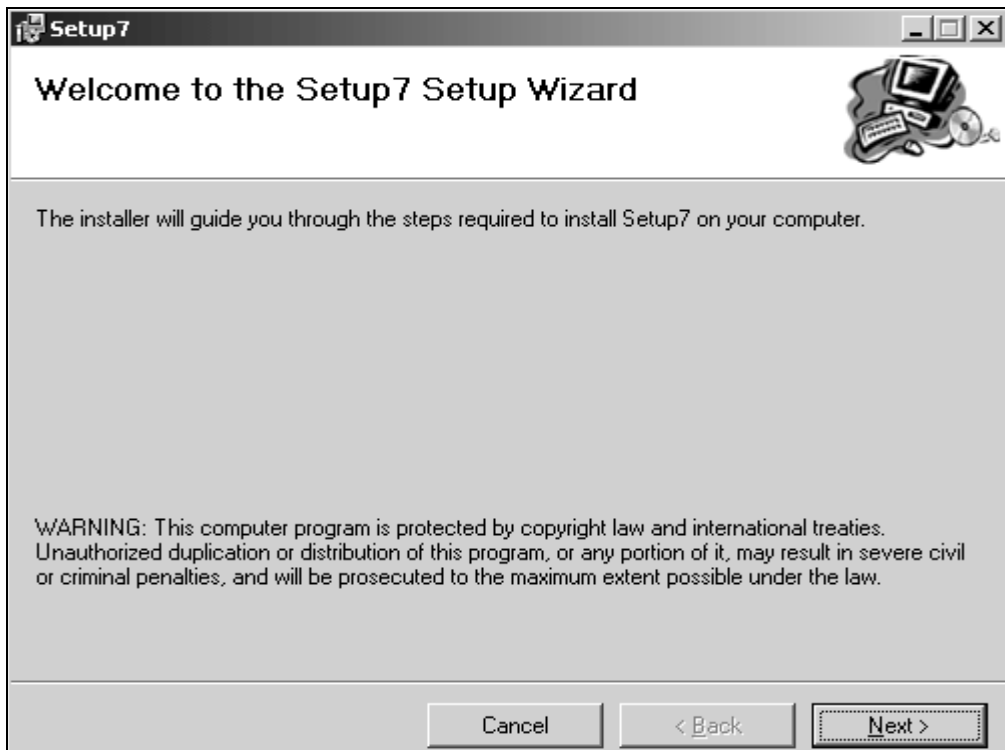


This window allows us to create shortcuts to the application on the desktop and in our Programs Menu. To create a shortcut, right-click on the folder *"User's Desktop"* and select *"Create Shortcut to User's Desktop"*. Rename the shortcut to *"Deployment"*. If we want a shortcut to the application from our Programs Menu, right-click on *"User's Program Menu"* folder and select *"Create Shortcut to User's Program Menu"*. Rename the shortcut to *"Deployment"*. Once you are finished with it, click on *"Application Folder"* and open its properties. In the properties window set the property *"Always Create"* to True. Set the same for *"User's Desktop"* and *"User's Programs Menu"* folders. If you want any additional information to include with the set-up project, like the manufacturer, author etc, click on Setup1 project and open its properties. You can set additional information here. Once you are done with it build the project by right-clicking on Setup1 and selecting *Build*. This builds the application. The setup file is created in the debug folder of Setup1 project.

Chapter 17

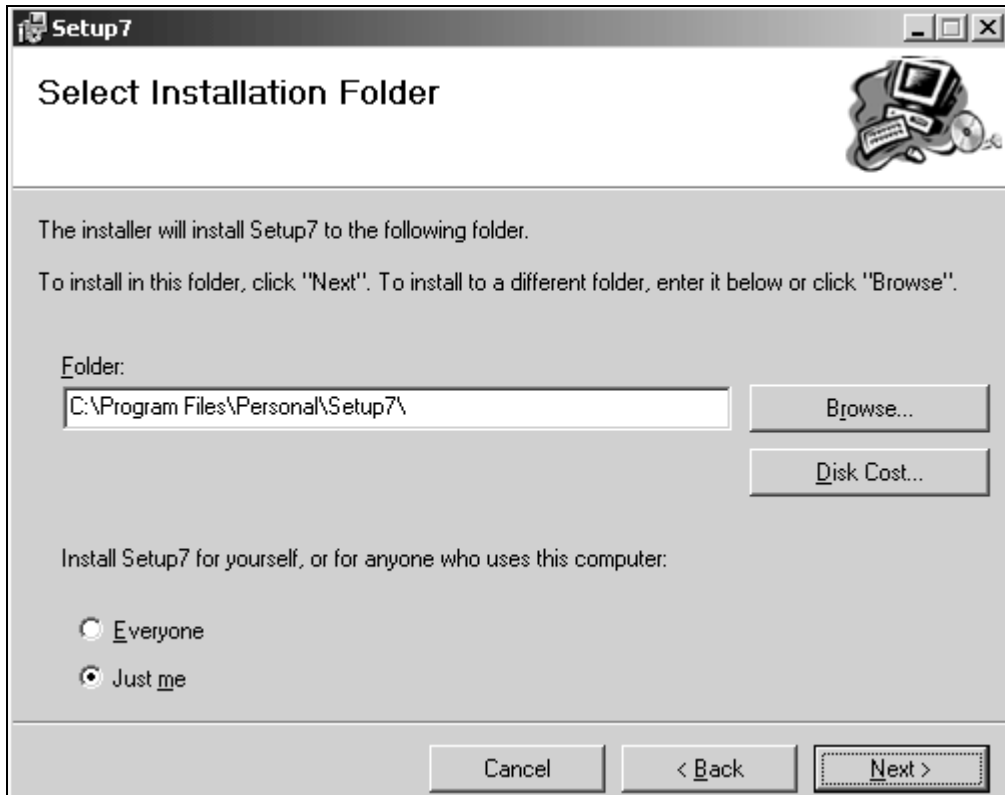
Deploying the Application

To deploy the application we need to copy Setup1.msi file to the target machine. Once copying is done, double-click that file which opens the Windows Installer which is a new window which says "Welcome to Setup1 Setup Wizard". It looks like the image below (mine was Setup7, yours will be Setup1).



Click next to move to next window which allows us to specify the location where the application should be installed. It looks like the image below.

Coalesce



Select the location for installation and click next. Clicking next installs the application. The confirmation window looks like the image below.

Coalesce



Now, double-click the newly installed Deployment.exe file to run and to get the desired result. You can select that from your Programs Menu or Desktop. That completes the process of Deploying Applications.

Make sure the Target Machine on which the application will be installed supports Windows Installer and .NET Framework.

Debugging Applications

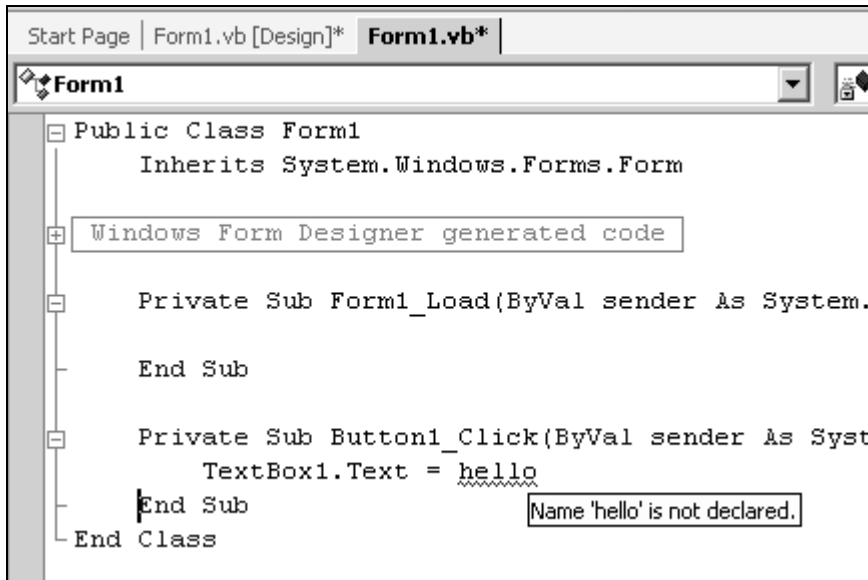
In this section we will focus on some common issues while debugging our applications and we will take a close look at the *Debug* Menu Visual Studio .NET provides us. Errors in programming are inevitable. Everyone of us introduce errors while coding. Errors are normally referred to as bugs. The process of going through the code to identify the cause of errors and fixing them is called Debugging.

Types of Errors

Syntax Errors

The most common type of errors, syntax errors, occur when the compiler cannot compile the code. Syntax errors occur when keywords are typed incorrectly or an incorrect construct is parsed. The image below displays a syntax error in the code editor.

Coalesce



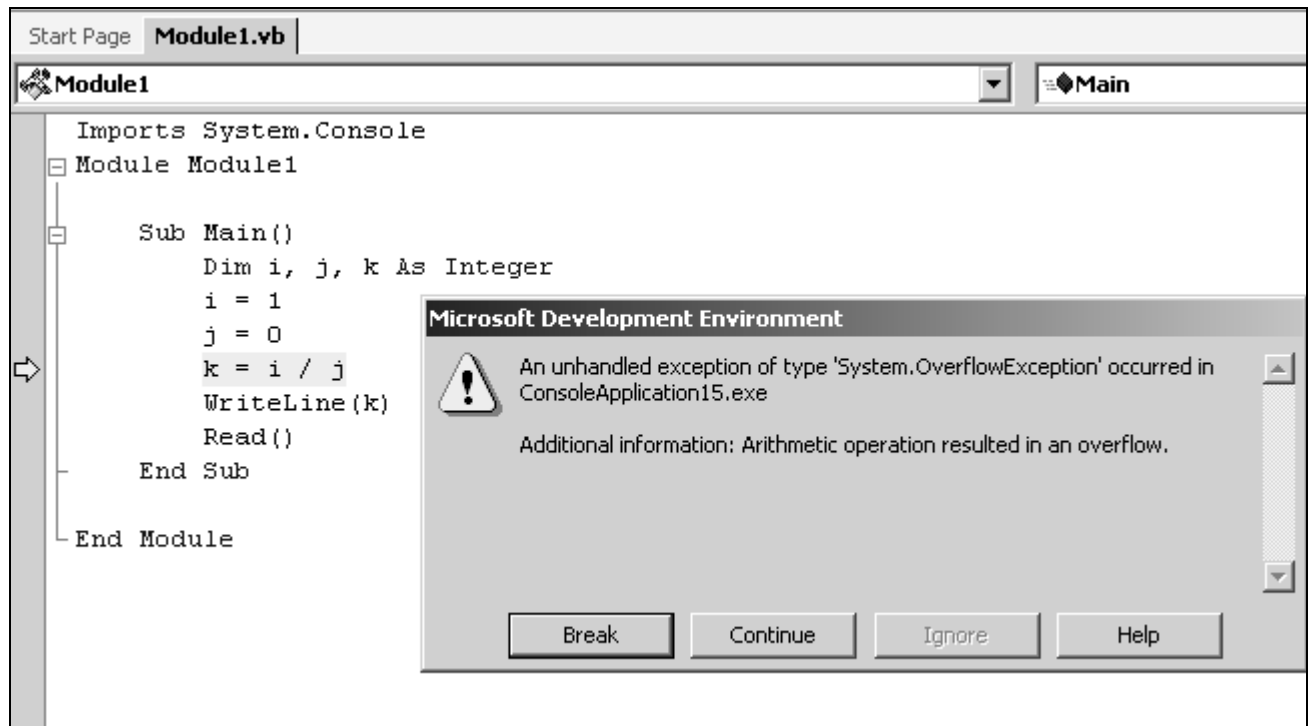
Fixing Syntax Errors

VS .NET allows us to easily identify syntax errors. When we build the project, syntax errors are automatically detected and underlined in code. Errors that are detected are also added to the Task List window. You can double-click any error in the Task List window and the cursor will immediately highlight that error in the code window. In most cases this is sufficient to correct the errors.

Run-Time Errors

Run-Time errors occur when the application attempts to perform a task that is not allowed. This includes tasks that are impossible to carry out, such as dividing by zero, etc. When a run-time error occurs, an exception describing the error is thrown. Exceptions are special classes that are used to communicate error states between different parts of the application. To handle run-time errors, we need to write code (normally, using Try...Catch...Finally) so that they don't halt execution of our application. The image below shows a run time error.

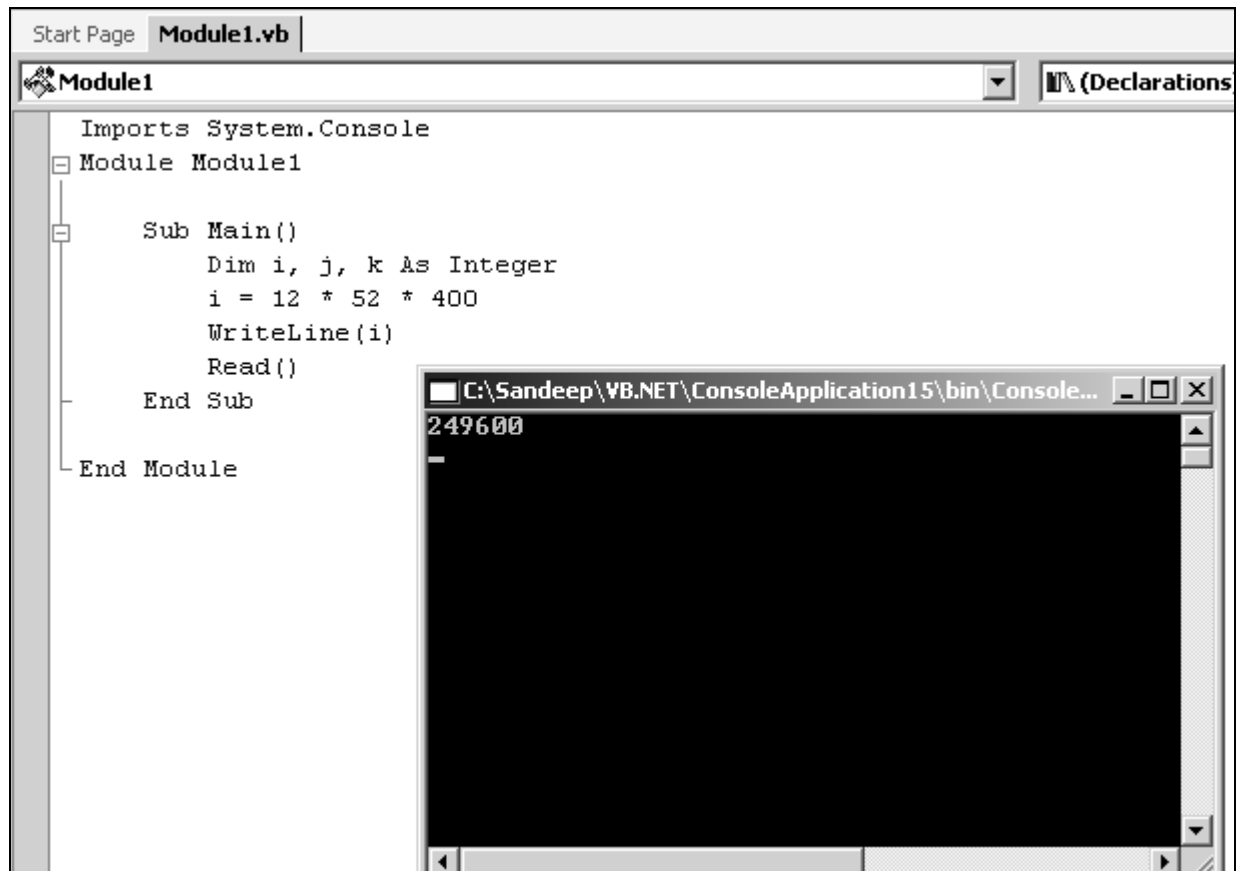
Coalesce



Logical Errors

Logical Errors occur when the application compiles and executes correctly, but does not output the desired results. These are the most difficult types of errors to trace because there might be no indications about the source of the error. The only way to detect and correct logical errors is by feeding test data into the application and analyzing the results. The image below describes a common scenario where logical errors occur.

Coalesce



Assume, the image above performs some calculations for the purpose of taxation. It multiplies three numbers. First number is the hourly wage an employee gets, second number is the number of weeks in a year and third number is the number of hours a week (40 hrs fulltime) the employee works. What happens here is, the code still executes correctly and produces the output but the produced output is not the desired result as the third number in code is 400 instead of 40. The output shown is 249 grand instead of 24 grand. This is a simple scenario where logical errors occur.

This section focuses on Breakpoints and the Debug Menu item found in Visual Studio .NET, the items within it and their usage.

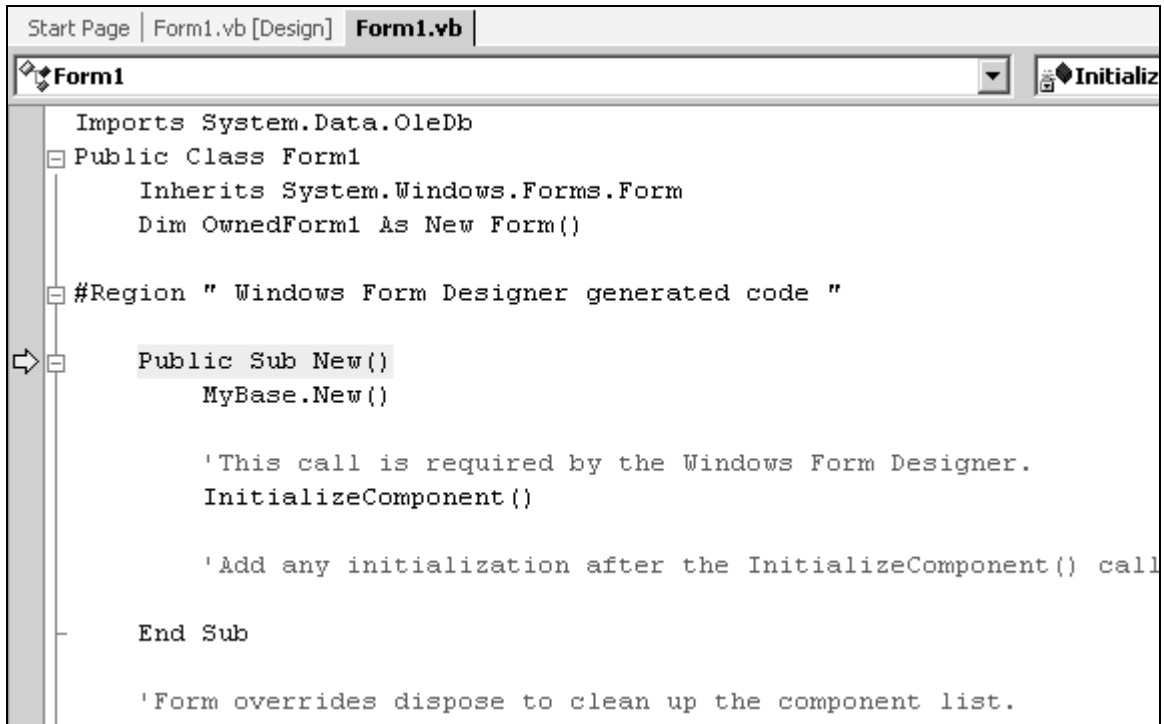
Break Mode

Break Mode allows us to halt code execution and execute code one line at a time. In break mode, we can examine the values of application variables and properties. Visual Studio .NET enters break mode under any of the following circumstances:

- When we choose Step Into, Step Over or Step out from the Debug menu
- Execution reaches a line that contains an enabled breakpoint
- Execution reaches a Stop statement
- An unhandled exception is thrown

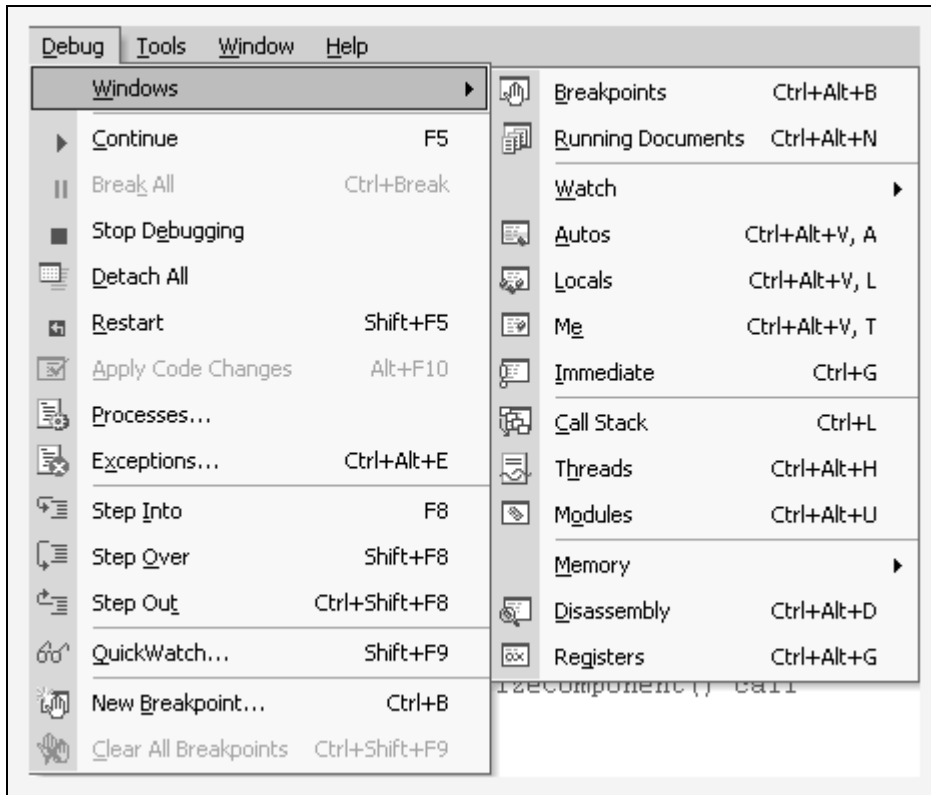
Coalesce

The image below shows the code designer window in Break Mode.



Once in Break mode, we can use the debugging tools to examine our code. The image below shows the features on Debug menu.

Coalesce



Debug Menu Items from the above image summarized as follows:

Windows: Opens a submenu that allows us to choose a debugging window to view.

Start/Continue: Runs the application in debug mode. In Break mode, this option continues program execution.

Break All: Causes program execution to halt and enter break mode at the current program line. You can choose continue to resume execution.

Stop Debugging: Stops the debugger and return to design mode in Visual Studio.

Detach All: Detaches the debugger from all processes it is debugging. This closes the debugger without halting program execution.

Restart: Terminates and restarts application execution.

Apply Code Changes: Works for C/C++ programming. Doesn't work for VB and C#.

Processes: Displays the processes window.

Exceptions: Displays the exception window.

Step Into: Runs the next executable line of code. If the next line calls a method, Step Into stops at the beginning of that method.

Step Over: Runs the next executable line of code. If the next line calls a method, Step Over executes that method and stops at the next line within the current method.

QuickWatch: Displays the QuickWatch window.

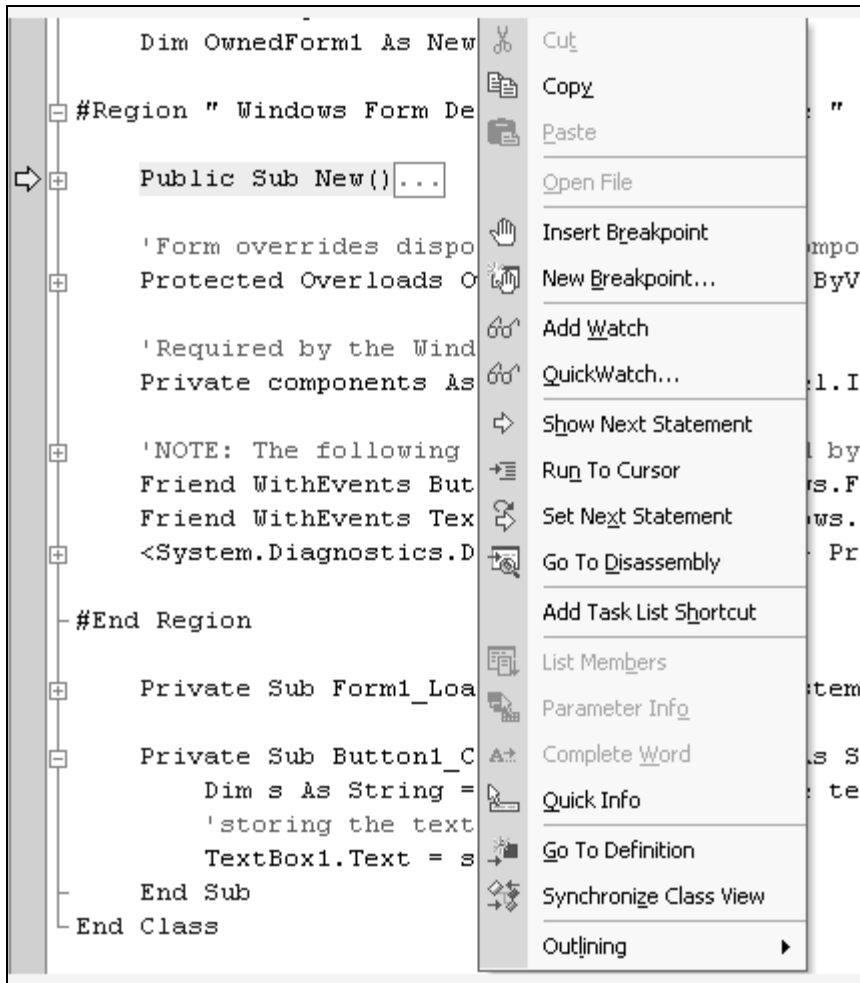
New Breakpoint: Displays the New Breakpoint Window.

Clear All Breakpoints: Removes all breakpoints from the application.

Disable All Breakpoints: Disables all breakpoints, but does not delete them.

There are some more debugging functions that can be accessible by right-clicking on an element in the code window and choosing a function from the pop-up menu. The image below displays those functions and summarizes them.

Coalesce



Insert Breakpoint: Inserts a breakpoint at the selected line.

New Breakpoint: Displays the new Breakpoint window. Identical to the Debug menu item of the same name.

Add Watch: Adds the selected expression to the watch window.

QuickWatch: Displays the QuickWatch window.

Show Next Statement: Highlights the next statement to be executed.

Run To Cursor: Runs program execution to the selected line.

Set Next Statement: Designates the selected line as the next line of code to be executed. The selected line should be in the current procedure.

Breakpoints

We can insert breakpoints at lines of code that will always cause the application to break in the debugger. Breakpoints are used to set lines of code that will halt code execution. There are four types of breakpoints as discussed below:

Function breakpoints: Causes the application to enter Break mode when the specified location in the function is reached.

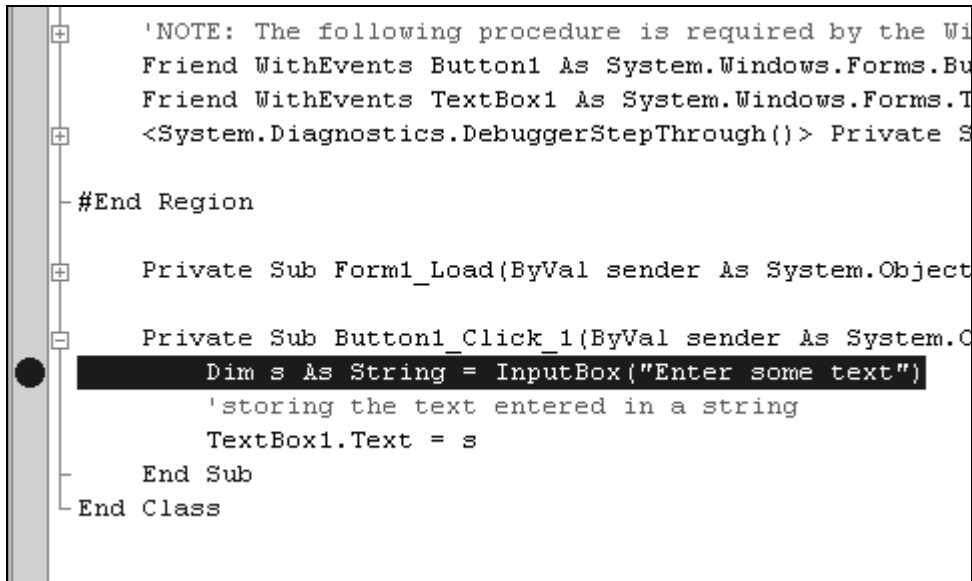
File breakpoints: Causes the application to enter Break mode when the specified location in the file is reached.

Coalesce

Address breakpoints: Causes the application to enter Break mode when the specified memory address is reached

Data breakpoints: Causes the application to enter Break mode when the value of a variable changes. Not available in VB and C#.

The image below displays the code designer when a breakpoint is inserted.

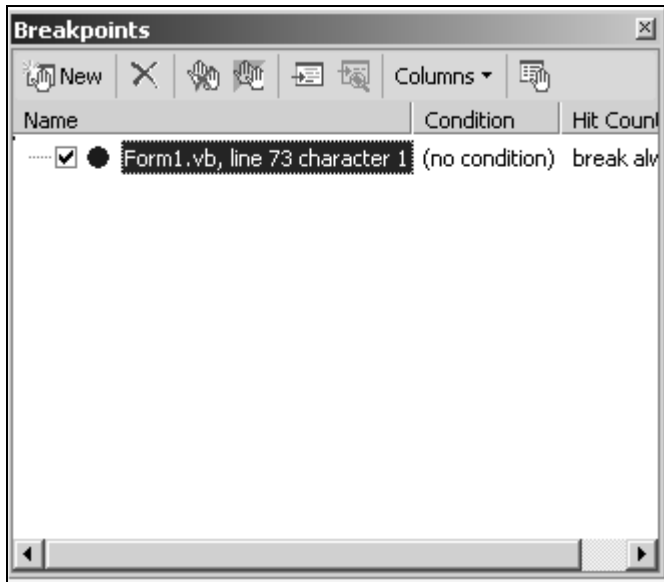


When setting breakpoints with the New Breakpoint window, we can attach conditions that determine whether or not execution halts when the breakpoint is reached. Pressing the Condition button displays the Breakpoint Condition window which allows us to designate an expression and causes the breakpoint to be active only if the breakpoint is true or if the expression changes.

Breakpoints Window

The Breakpoints Window lists all of the breakpoints in the project like where the breakpoint is located and any conditions attached to it. We can disable a breakpoint by clearing the check box next to it. The button in the Breakpoints window allows us to create a new breakpoint, delete a breakpoint or clear or disable all breakpoints. The Columns drop-down menu allows us to choose additional columns of information about the breakpoints. The image below displays a breakpoints window.

Coalesce



Removing and Disabling Breakpoints

We can remove a breakpoint by clicking on the circle like display towards the left of the code window where the breakpoint is inserted. To disable a breakpoint, right-click the breakpoint in the code editor and select "Disable Breakpoint".

