Supplemental Readings For the Express Edition Videos

To Be Used With

“Beginning Visual Basic 2005 Express Edition Video Series”

&

“Beginning Visual C# 2005 Express Edition Video Series”

Which you can download from:

http://lab.msdn.microsoft.com/express/beginner/

The videos and this document are presented by:

Bob Tabor, Visual C# MVP, LearnVisualStudio.NET

http://www.LearnVisualStudio.NET

Document Version 1.0
7/27/2005

About This Document

This document is a collection of articles I've written and compiled that should introduce the absolute beginner to basic concepts and help them more fully understand some of the complex ideas that are discussed in the Express Edition videos. After reading the first chapter "What is Computer Programming?" you should start watching the videos that accompany this document and from that point on, use this document as a reference when asked in a particular video to read more about a given topic.

I hope you enjoy the videos and this document. When you get a chance, please visit http://www.learnvisualstudio.net for more great content like this to help you get the most out of the new Express Edition tools.

Since you're reading this document, I'll assume you know absolutely nothing about programming. If that is the case, then you are in the right spot. So let's start from the beginning.

Sincerely,
Bob Tabor
LearnVisualStudio.NET

Table of Contents

What is Computer Programming?

Introduction

A computer, by itself, knows absolutely nothing. It doesn't even know how to display information on a screen or access a hard drive for data. It doesn't know how to access the Internet nor does it know how to play a sound. Despite popular belief, computers know absolutely nothing.

But computers are very good at following and repeating instructions flawlessly and very quickly. This is what makes these machines so powerful.

Someone had to write the instructions, called code, to make the computer do everything that we take for granted. Your ability to view this document and my ability to compose it alone took millions of instructions.

Computer programming is the act of writing code that is interpreted by the computer to carry out some repeatable task.

Despite popular belief, computers do not make mistakes (at least, not unless there is a physical problem with the computer.) Computers always do exactly what they are told to do. However, programmers don't always instruct the computer correctly. When this occurs, it is referred to as a bug. A bug can have varying degrees of severity. Some bugs can cause the computer to become unresponsive and others are mere annoyances.

There are many reasons why bugs are introduced into software, such as laziness on the part of the programmer, a misunderstanding of a particular feature of a programming language, or just not enough time to test every feature thoroughly. One way to write solid code is to understand the fundamentals and follow "best practices" – which are conclusions that other programmers have come to after years of experience.

Purpose of Computer Languages

Amazingly, everything that happens within a computer is a result of tens of thousands of tiny on and off switches.

- The color of every dot on your screen is a result of the processor determining what color should be there …
- Copying text to the Windows clipboard ...
- Opening a music file and listening to it ...

Tens of thousands of little switches have to turn on and off to route information to memory, to send signals to the video card or sound card, or to retrieve every bit of data from the hard drive. Fortunately, you don't have to code in 1's and 0's or rather, ON's and OFF's.

A computer programming language provides a more human-friendly means of communication with the computer. In some cases, while not exactly English-like, the syntax of a programming language is still a marked improvement over speaking to a computer in its native language!

The job of the computer programmer is to feed instructions to the computer using a computer language.

Compiling Code

Once a programmer finishes writing code, it must be compiled. Compiling is the process of converting the lines of code written in a computer language into the native language of a computer program – something it can understand. To compile the code, you need a compiler, which is a special piece of software that converts the source code you create into code that the computer can understand. It also performs other operations, such as checking to make sure there are no logic errors in your code, or syntax errors (which means you may have spelled something incorrectly.)

In modern programming language such as Visual Basic or C#, there are actually a few steps in between writing code and the physical computer receiving instructions. These languages are called "interpreted languages" because they are compiled into an intermediate language that is then interpreted by each type of computer they reside on. They are interpreted by a program called a Virtual Machine which takes the compiled intermediate code and executes it on a particular computer. For example, Java's claim to fame is the slogan: "Write once, run anywhere." Java has a separate Virtual Machine that runs on Windows, on Mac, Unix and Linux. This allows a developer to create code once and be assured that the Virtual Machine will determine how to work with the idiosyncrasies of each operating system and computer hardware. Since we are working with Visual Basic and C#, the Virtual Machine is actually called the .NET Runtime.

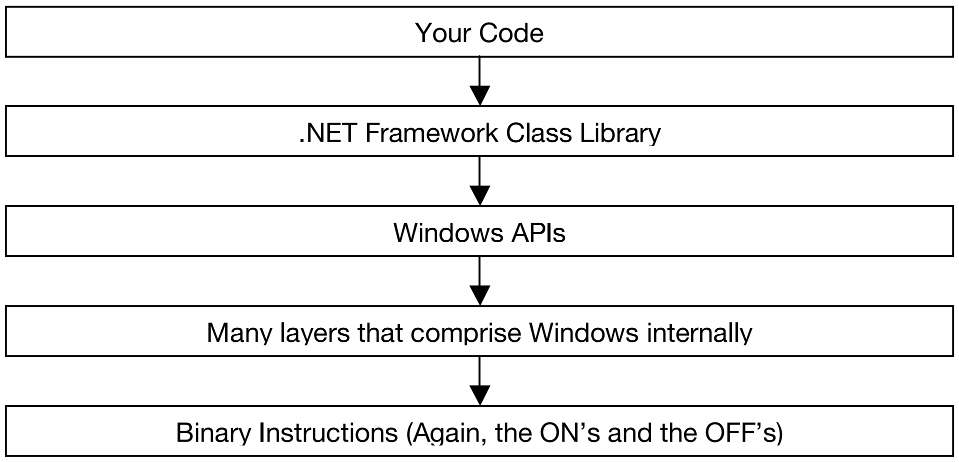
When you compile your Visual Basic or C# code, it is compiled into a different language called the MSIL, or rather Microsoft Intermediate Language. The compiled code will then reside in a file called an Assembly. Assemblies have an .exe or a .dll file extension. If you were to execute your assembly, it immediately attempts to load the .NET Runtime and the .NET Runtime then becomes responsible for executing the MSIL code.

Software Layers

A software layer is existing code created by other software developers that provide functionality that you, when you sit down to write applications, can use. Imagine how much code you would have to write to ensure that your application runs correctly on Windows 98, Windows ME, Windows NT, Windows 2000, Windows XP (both Home and Professional)! In Visual Basic and C#, you can use a library of code called the .NET Framework Class Library, or just the .NET Framework for short. This contains thousands of methods for easily performing thousands of tasks, such as:

- Working with Date or Time information
- Writing information to a file
- Reading from a database
- Connecting to the internet
- Displaying a window on the screen

The following is a very simplified illustration, but consider just a few of the layers that you will be building on top of:


Your Code
.NET Framework Class Library
Windows APIs
Many layers that comprise Windows internally
Binary Instructions (Again, the ON’s and the OFF’s)

Modern Programming Languages

Visual Basic and C# examples of languages you can use to create software for the Windows operating system. Other languages include: C, C++, Java, Delphi … and I'm probably forgetting a few.

Why, then, focus on Visual Basic or C# when there are so many other choices?

Some of the benefits of Visual Basic include:

- It is a very English-like programming language – more so than the other programming languages, its easy to learn for someone who has never used a programming language before.
- It is a ".NET Language", meaning that you can take advantage of all the benefits of the .NET Framework Class Library, the .NET Runtime, etc. We've only reviewed a few of those benefits above, but there are many more and we'll discuss them throughout this series of video lessons.
- There are plenty of community resources on the World Wide Web for Visual Basic. There are also many books on Visual Basic topics, and many developers who you can find who know it if you have questions.
- Lots of opportunities for Visual Basic programmers as businesses seek to convert Visual Basic 6 code into .NET applications using Visual Basic 2005.

The downside to Visual Basic is:

- Its less C-Style Syntax than C#, which means that learning additional programming languages such as Java, C#, C++ and others might be more difficult since there are fewer similarities between VB and the others.

If you are committed to learning a programming language, Visual Basic is a good place to start if you have no prior programming background. C# resembles so many other languages that if you have some experience in C, Perl, Java, etc. you may prefer it.

Personally, I use them both just to keep my programming skills sharp in both languages.

Some benefits of C# include:

- It uses a C-style syntax, which means that learning additional programming languages such as Java, JavaScript, C++ -- or even C itself – becomes a little easier. Very few programmers today know just one programming language, so by learning C# as your first language gives you an advantage when getting started.

- It is a ".NET Language", meaning that you can take advantage of all the benefits of the .NET Framework Class Library, the .NET Runtime, etc. We've only reviewed a few of those benefits above, but there are many more and we'll discuss them throughout this series of video lessons.

- There are plenty of community resources on the World Wide Web for C#. There are also many books on C# topics, and many developers who you can find who know it if you have questions. In fact, many of the better books are written originally in C#, and at a later point are translated into Visual Basic.NET.

- Lots of opportunities for C# programmers as businesses seek to convert Visual Basic 6 code into .NET applications using C#. In many cases, Visual Basic programmers are "jumping ship" to learn this new language because of the benefits listed above.

The downside to C# is:

- It is NOT a very English-like programming language – this is why so many people start with Visual Basic instead of C or C++ or Java. The C style syntax is not as easy as the BASIC syntax (IMHO) to learn for someone who has never used a programming language before.

If you are committed to learning a programming language, C# is a good place to start. In my personal opinion, if you are considering whether to start with Visual Basic or C#, C# is the better language. The main reason I say this is because it doesn't have to support "legacy" code, meaning that the C# language is more streamlined than Visual Basic, which was forced to support many of the older features of the language due to the outcry of some programmers who did not want to re-write large amounts of code in order to take advantage of the .NET Framework.

Additionally, if you review the job listings on Dice.com, Hotjobs.com, etc. C# developers (especially **experienced** C# developers) are in high demand and a survey by Visual Studio.NET Magazine found that C# developers have higher incomes than Visual Basic developers, all things being equal (i.e., job description, experience, education, etc.)

Software Development Goals

Having made the decision to learn Visual Basic or C#, there are some principles and goals that each developer must be absolutely committed to. These goals permeate each project you'll ever work on, and will distinguish you as a developer who takes pride in his workmanship. They are as follows:

Create software that fulfills requirements and is accurate – If you are building software for someone else, you must make sure that you give them exactly what they have specified, despite how you may feel about their requests. It is not your job to demand that software operate a certain way, or that the business changes how they work to accommodate your application. This is not to say that you can not present alternatives based on your experience and how you feel the application could be improved. Too many times lazy developers have tried to talk managers or clients out of a particular feature because they knew it would be too hard to develop in the manner the customer requested. My advice: "Get over it!" Force yourself to learn something new if necessary. Deliver software that fulfills the requirements that were handed you and ensure that what you've created accurately performs the requested functionality.

Write software that has virtually no bugs – There is a concept in programming called "Good Enough Software". This means that you build software the best you can knowing full well that it has problems, but it's the best you can do in the time allotted. I have mixed feelings about this. On the one hand, I sympathize with developers who are under a tight timeframe and may not have enough time or resources to develop completely bug free applications. On the other hand, I know the headaches that come from being "haunted" by code that I've written just "good enough" to be sent to a client, only to have the client express their dismay at the problems they experience with the software. I speak from experience: this is NOT a good feeling. Obviously a balance much be reached (there has never in the history of programming been a completely bug free application), however I would err on the side of perfection. If nothing else, you can gain a reputation of building solid software rather than a reputation of shoddy work. Ideally, you could develop solid software AND meet deadlines, and the way you accomplish that is by knowing well the fundamentals of application architecture and abiding by best practices.

Create software that is usable – Usability has become a focal point of software and web site development because the impact of a bad user interface design can cripple the effectiveness of the software. The way you create usable software is by learning the fundamentals of interface design and by staying true to best practices.

Create software that performs well – Commit yourself to learning how to write code that is efficient, learning how to optimize the code you've created. Writing optimized code is a lifelong quest, researching the impact of each feature of a programming language, but by adopting best practices and knowing the fundamentals of writing and structuring your code, you can create applications that perform well.

There are two themes that I've highlighted throughout this section:

1. Knowing the fundamentals
2. Following Best Practices

Throughout this series of video lessons, I'll be focusing in on these two concepts to ensure that you learn correctly from the start. I personally feel that too many developers want to use the latest complex features or techniques of a programming language before they are intimately familiar with the basics of that language. The result is predictable, as the old saying goes "A little knowledge is a dangerous thing." Studying and getting the fundamentals right is a basic truism of life, of football, and of programming.

Also, too many people try to pave their own way. I've been on multiple projects where people tried to use a non-standard approach to a particular problem. It wasn't always easy for me to articulate exactly why I was uncomfortable with that approach, but what I did know was that it simply "didn't feel right." Sure enough, about two months into development on a software project and the developer would realize that the decisions he made earlier now has hemmed him into a corner. He would have to re-write the code he created, or worse, just try to "hack" a solution, which forces him to sacrifice his ability to deliver an accurate, bug-free, usable, well performing application to the customer. This is why following best practices is so important – why make your own mistakes when you can learn from the mistakes of others?

Decide now that you will commit to coding correctly from day one.

Learning Visual Basic or C#

Here are some tips on learning Visual Basic or C#:

You learn by doing. As you watch the videos in these lessons, try to follow what I've done and re-create the project and the code. Pause the video. Rewind it and watch it over. Don't just mindlessly copy source code from a website or from these videos. If something is not completely clear, take the time to use the Help feature to do some extra research.

Be patient. I began to teach myself Visual Basic 3 back in June 1993. I didn't get my first real job using Visual Basic until February 1995. It took a lot of nights and weekends to learn it well enough to get a job. It probably won't take you quite as long … after all, I didn't have videos like these to get me started. ☺

How long will it take you to learn? That depends on your level of commitment and your background in software development. However, after going through these lessons and watching the videos in this series, as well as all the videos on LearnVisualStudio.NET, you should have a pretty rich understanding of how to develop software using Visual Basic and C# and the Visual Studio.NET environment.

Find a project to do for a friend, your church, a small local business or a department within your company. If you have a project in mind as you begin to learn, it will force you to learn things you otherwise might overlook. Once you are finished, you will inevitably feel ambivalent: you'll be so proud of what you've accomplished, and at the same time you will look back and realize how utterly pathetic that first attempt at programming was. Although I'm not a golfer, I understand there are similarities. If you are competitive and passionate, you'll constantly seek ways to improve your game. But even Tiger Woods is never completely satisfied, although he may be one of the best who has ever played the game.

Obtaining the Tools You'll Need

If you haven't done so already, download Visual C# 2005 Express Edition or Visual Basic 2005 Express Edition from:

http://msdn.microsoft.com/express/

The installation process is fairly straight forward ... if you encounter something you don't understand (which I doubt will happen) just accept the defaults for the installer.

Once installed you'll be ready to continue reading in this document, but a better idea might be to start viewing the videos lessons that you have downloaded from:

http://lab.msdn.microsoft.com/express/beginner/

At various points during the videos you'll be instructed to read more about a given topic in this document.

Object Oriented Programming Using Visual Basic 2005

Introduction

This chapter accompanies video 6, “Object Oriented Programming Fundamentals”.

Object Oriented Programming seeks to reduce the complexity of creating large applications by breaking the application down into smaller, manageable classes of code. Each class represents an idea, whether tangible and concrete (such as a Product or Employee) or conceptual (such as Inventory or Order).

In the past, most programmers wrote procedural code, which was characterized by global variables (variables that could be used at any time anywhere in your application), many modules that contained inter-dependencies, and long code passages that contained many “goto” statements that jumped all over the place (often referred to as “Spaghetti code”). While procedural code is not inherently evil, carelessness caused problems as developers tried to fix bugs … changing a single line of code would have major devastating consequences in other parts of the application because the entire application was so interdependent. To make changes, programmers had to re-write entire applications because it was less work than trying to untangle the mess of code that they were left with.

Object Oriented Analysis, Design and Programming sought to improve on the entire software development process by taking a simplified approach to application design that more closely resembled everyday life.

You won't get very far in modern software development before hearing about Object Oriented Programming. It’s also known as OO for Object Oriented, and the purpose of OO programming is to reduce the complexity involved with writing and maintaining applications.

In real life, everything around us has properties and methods, although we generally don't think about life in these terms. For example, a Pencil has a color, a length, a lead quality ("bring your #2 pencils") ... and it also has methods, or things it can do like Write and Erase.

The basic idea behind object oriented programming is that if we can write code that mimics real world objects, like pencils, cars, customers, payments, etc. then it will make programming a lot easier. This is a gross over-simplification however it's a starting point.

Consider a car as an example of a real-world object oriented system that is comprised of many discrete parts. The car itself is an object. The engine inside the car is an object. The exhaust system is an object that contains even more objects. The engine depends on the exhaust system, but it has no idea how the exhaust system works. Even though the car parts don't know much about each other – except how to connect to each other – amazingly the parts all do their individual jobs and the end result is a working machine. If one part goes bad, it can be replaced without adversely affecting the rest of the car.

Additionally, you don't have to know anything about how a car works in order to drive it. You sit down in a comfortable seat, turn on the ignition key, and operate the steering wheel, the gas pedal, the brake and occasionally a turn signal. However, your instructions trigger hundreds of individual parts without you having to worry about them. This focuses on a major OO concept called Encapsulation. All of the powerful engine and navigation components are encapsulated behind two or three controls that you (the user) work with.

To extend this example further, there are specifications that define what a car is in general – if not in reality, then at least in our minds -- and each automaker inherits and extends those specifications to give their car a unique look, performance benefit or price benefit. In other words, each automaker inherits from the ideal "car" to create a more specialized car. Still, from a programmer's perspective, the Hyundai and the Mercedes have more in common than they have differences. The same general engine design, exhaust system design, number of tires, operation of the steering wheel, seats, pedals, turning signal, etc. exist in both cars. They both borrow from the design and mechanics of how a car works. In software terms, this is referred to as Inheritance, another important concept in Object Oriented Programming.

Also, all cars can go to the same gas stations and fill up using the same gas pumps, despite the fact that a Mercedes may cost

$60,000 more than a Hyundai. This is known as Polymorphism, meaning that even though there are different objects, you can interact with them the same way because they both derive from the ideal car.

So, when defined in this way, I hope you can begin to see the value of looking at software the way we analyze the world around us.

For this lesson, we'll focus on the most basic building block of OO programming, and that is the Class. A class is simply a blueprint. The Class defines the fields, properties, methods and events that an object will have.

Fields? Properties? Methods? Don't worry … we'll talk about what these are in a moment.

A class is a blueprint only; its only job is to define what data it should store (the variables, or rather Fields) and what actions the object should take (the procedures, or rather the Properties and Methods). But since a class is just a blueprint, it is not the ACTUAL object … just a definition of the object's appearance, behaviors, etc.

Just like in the real world, you take a blueprint to a contractor, or take the blueprint to a baker and they create an instance of that blueprint, resulting in a house or a cookie respectively. In our case, we'll create an instance of the class. The instance is a manifestation of that class, which we then call an Object. So let's define a class:

Visual Basic:

```
Class Car

End Class
```

This class does almost nothing. So let's add some properties and methods.

Visual Basic:

```
Public Class Car
      Private m_make as string
      Private m_model as string

      Public Property Make() as string
            Get() as string
                  return m_make
            End Get
            Set(ByVal Value as string)
                  m_make = Value
            End Set
      End Property

      Public Function Drive() as string
            if make = "Oldsmobile" then
                  return "Chicago"
            else
                  return "Toledo"
            end if
        End Sub
End Class
```

Admittedly, this class doesn't do much but will allow us to see what a real Class looks like. First, it has two private fields and a public function. We talked briefly about scope in the videos, and the words "Private" and "Public" have special meaning in terms of Scope. I'll have more to say about it in later in this chapter when we look at Properties, Fields and Methods.

Before we can interact with the Make property or the Drive function, we must create an instance of the Car class; or rather, we must instantiate the Car object.

Visual Basic

```
Dim myCar as Car
myCar = New Car
Dim location as string
myCar.Make = "Oldsmobile"
myLocation = myCar.Drive()
```

The most important part of this example is the first two lines of code. First of all we dimension a variable myCar that is of type Car. If you remember back to video lesson 4 we worked with Primitive Types like strings and integers. Now we are working with a Complex Type, one that we created ourselves. Its “complex” because it can contain multiple values (Make and Model) and perform actions (like the Drive method). So when we talk about “types” we are talking about “classes”… as we learn more about .NET, we’ll see that even Primitive Types are defined as classes which have methods and properties.

The second line of code is where the magic actually happens. In line 1 we created a variable of type Car, but as of yet it is not actually a Car. At the moment we use the “New” keyword, a Car object is created in memory and we store a reference to that new Car in the myCar variable. Let me clarify this. All the information about the newly created Car object is stored in your computer’s memory. The variable myCar is given the address of that memory space, which is called a “reference”. Now, whenever you use the word “myCar” in code, it just refers the compiler to the memory address that represents that instance of the Car object.

There's a subtle difference between dimensioning a variable as a type (or class) and actually creating an instance of that type (or class). This will become more apparent as we talk about Class Constructors near the end of this chapter.

Almost everything you work with in VB, whether you know it or not, is a Class. Just take a look back at all the forms we’ve created in the previous video lessons.

Fields

Let’s look at the Car class we created:

Visual Basic:

```
Public Class Car
      Private m_make as string
      Private m_model as string

      Public Property Make() as string
            Get()
                  return m_make
            End Get
            Set(ByVal Value as string)
                  m_make = Value
            End Set
      End Property

      Public Function Drive() as string
            if m_make = "Oldsmobile" then
                  return "Chicago"
            else
                  return "Toledo"
            end if
      End Function
End Class
```

First lines of code are fields. Fields are simply variables defined within a class. They can be private like we have in the Car example, or they can be public.

Ideally, you should never use Public Fields, but rather use Public Properties instead. We'll talk about that more in a moment, but let’s add three more BEST PRACTICEs to our ongoing list:

BEST PRACTICE: Instead of Public Fields, use Public Properties. As a general rule, keep as many fields and methods private, and make them public only if there is a reason to do so.

BEST PRACTICE: use an m_ to denote a private field. If the field or method is private, use camel case.

BEST PRACTICE: Public fields, properties or methods should be in Pascal Case.

Camel Case == engineSize or elapsedMileage
Pascal Case == EngineSize or ElapsedMileage

Properties

Let’s add a public field for a moment just to show how we would reference that class variable in our code:

(This is just a re-cap of the Car class … all we’re doing is adding a Public field)

Visual Basic:

```
Public Class Car

      Public ElapsedMileage As Integer

End Class
```

(This code would be used to create an instance of the Car object and use the new Public field we just added)

Visual Basic:

```
Dim myCar As Car

If myCar.ElapsedMileage = 0 Then
      myCar.ElapsedMileage = 10000000
End If
```

This illustrates two ideas. First, it shows how to use the . (dot) syntax to access public fields and set or retrieve their values. Second, it shows the problem with Public Fields. You could set their values to nonsensical values. Very few cars have traveled 10 million miles. We would obviously like to limit that to a sane amount. We can do validation (also known as "sanity checks") on values, or filter the value of a Field by using a Property instead. Let’s change the ElapsedMileage from a Public Field to a private field:

Visual Basic:

```
Private m_elapsedMileage as integer
```

And then create a public property called ElapsedMileage:

Visual Basic:

```
Public Property ElapsedMileage() as integer
     Get () as integer
            return 5
     End Get
     Set (ByVal Value as integer)
      ' Do nothing
    End Set
End Property
```

First, did you notice how the IDE completed the structure of the Property statement for us? We can still modify it, but it is a nice convenience that the IDE gives us.

Second, notice that there are two distinct parts of the Property statement ... a Get and a Set. The Get retrieves and returns a value to the code that calls it. The Set assigns a Value. Note that the Property statement does nothing by itself; we have to decide which value to return when the Get statement is called and what to do with the Value that is passed into the parameter of the Set statement. We could ignore these values, "hard code" them, or whatever we like. Typically, we'll do this:

Visual Basic:

```
Public Property ElapsedMileage() as integer
     Get ()
          ' You could do some validation
          ' or modification here
          return m_ElapsedMileage
     End Get

     Set (ByVal Value as integer)
          ' You could do some validation
          ' or modification here
          if Value > 1000000 then
               m_ElapsedMileage = 1000000
          else
               m_ElapsedMileage = Value
          end if
     End Set
End Property
```

In this example, we chose to use a private field m_ElapsedMileage to store the parameter of the Set statement. However, we do a little due dilligence to make sure that someone didn't enter a bogus amount of miles. Then we use the m_ElapsedMileage again when someone calls the Property Get statement.

So now let’s see how these are used in code:

Visual Basic:

```
Dim myCar As Car

If myCar.ElapsedMileage = 0 Then
      myCar.ElapsedMileage = 10000000
End If
```

Yes, this code is correct. It is identical to the code we used to talk about Public Fields. You may wonder WHERE the call to the Get and Set statements is located. Actually, the calls are implied by the fact that we are reading the value in the “if ... then” statement, and the fact that we are assigning it a value in the myCar.ElapsedMileage = 10000000 statement.

If we add one more statement:

Visual Basic:

```
MessageBox.Show(myCar.ElapsedMileage.toString())
```

… what value do you think will be displayed? If you said 1 million you'd be correct. That is because of the way that we wrote our Property Set statement to filter out insane numbers like 10 million.

Methods (for those learning Visual Basic)

The last thing we'll talk about in this lesson is Methods, which is an object oriented way to refer to procedures (or functions). These terms are synonymous, but you'll sound cooler if you say Methods from now on rather than procedures or functions. There are still two different types of procedures -- I mean -- methods: there are Sub Procedures and Functions.

Visual Basic:

```
Public Function Drive() as string

      if m_make = "Oldsmobile" then
            return "Chicago"
      else
            return "Toledo"
      end if

End Function
```

Methods can be either Private or Public scope. Many times, you'll want Private methods to be used as helper functions or utilities that hide how the class does its work. Ideally, classes are like mysterious "black boxes" to the code that calls them. You know nothing about how the object works except for the Public properties and methods, and that is good. This is known as "Encapsulation" in Object Oriented programming terminology.

Overloaded Methods

Consider the following two Drive methods closely:

Visual Basic:

```
Public Function Drive() As String
      If m_make = "Oldsmobile" then
            Return "Chicago"
      Else
            Return "Toledo"
       End If
End Function

Public Function Drive(ByVal make As String) As String
      If make = "Oldsmobile" Then
            Return "Chicago"
      Else
            Return "Toledo"
      End If
End Function
```

This is called Overloading the Method. We've overloaded it with two different ways of calling the function. We can either call it passing in no parameters, or call it passing in the make. Each method returns a string, but the parameters and how the method works internally can be different. In order to overload a method, the method signature must be different. "Method Signature" is just a technical term that means each one must have a different set of parameters. Why would you overload a method? When you create classes, you don't always know how they might be used up front in your application. Sometimes you don't have all the data readily available to you that you would need to call a method, so you need options on how to call the method.

Introducing Constructors

First of all, classes have Constructors, which is a special method that allows you to initialize your fields, or whatever else you may want to do when your object is first created. In Visual Basic.NET, you create a Constructor using the New statement:

Visual Basic:

```
Public Sub New()
     m_make = "Unknown"
     m_model = "Unknown"
     m_elapsedMileage = 0
End Sub
```

Here we've set our private fields to default values. The Constructor gets called whenever the New keyword is used in association with our class, like so:

Visual Basic:

```
myCar = New Car
```

Also, the constructor is ALWAYS the first code to execute within your class, however having a Constructor in your class is optional. While Constructors are not required, it is a good idea to use constructors to initialize the private property (private member) values of your new object.

I've used the term "initialize" several times; let me explain what I mean. Initialization is the process of taking steps to ensure that your object will function properly by the time the object is used in an application. That means different things based on how you design your object. For example, you may choose to create a connection to a database, or create instances of child objects, or set variables (like private fields) to default values, etc. The constructor is the perfect place to execute code that MUST RUN in order for your object to perform correctly.

BEST PRACTICE: Use Constructors to initialize values for a newly created object.

Overloading Constructors

Since a Constructor is simply a method, you can overload the Constructor. Recall from earlier in this chapter that overloading a method means that you can have different implementations of your method based on a different method signature.
I could create another Constructor that allows me to initialize the values as soon as I create an instance of the class:

Visual Basic:

```
Public Sub New(ByVal make As String, _
            ByVal model As String, _
            ByVal elapsedMileage As Integer)
      m_make = make
      m_model = model
      m_elapsedMileage = elapsedMileage
End Sub
```

Then, this is how I would use that constructor when I create a new instance of the class:

Visual Basic:

```
myCar = New Car("Nissan", "Altima", 31000)
```

Death of an Object

So we understand what happens when we create an object, but what happens when we are finished using an object? This isn't an easy answer, so try to understand what I'm about to say involves some concepts that are rather advanced and I'm not telling you everything.

We sometimes talk about things being in a plastic bubble, maybe you've seen the movie "The Boy in the Plastic Bubble" or watched the Sienfeld episode with the "Bubble Boy". The bubble protects people with weakened immune systems from the outside world. The Bubble is a totally controlled environment. In a like manner, that is what the .NET Runtime does for your applications. It protects your application from the outside world, so that other programs can't corrupt its memory space, and it cleans up after your program when it's finished. Protecting an application's allocated memory prevents memory leaks or other bugs from shutting down your application immediately. To guard against this, the .NET Runtime has a Garbage Collector, which is a process within the .NET Runtime that searches for object references in your code that are no longer needed. When it finds an unused object it disposes of it, or rather, it removes the object from memory.

As your code is executing, the .NET Runtime will flag those objects that it knows it won't need again, and after a while the Garbage Collector comes through and disposes of all the objects that are flagged for destruction. Right before the Garbage Collector destroys your object, you will get a last second chance to execute some code, which is called a Destructor … I'll cover that in the next section.

Many developers try to help the .NET Runtime by doing the following:

Visual Basic:

```
myCar = Nothing
```

However this is not necessary. Older versions of Visual Basic required you to do this, but you don't have to anymore. When a given procedure finishes processing, the variables that were defined in that procedure go "out of scope" which means that their values are no longer available -- the task is complete and the variables are no longer needed. As the .NET Runtime executes, it looks ahead and decides when it needs to destroy an object, based on whether or not the object's reference is used in future code.

So, by waiting until the very end of the procedure to add this code (as is a common practice in previous versions of Visual Basic):

```
myCar = Nothing
```

… you might actually be hindering rather than helping the .NET Runtime to clean up memory sooner. Although, lets be honest, it really won't make that big of a difference in smaller programs. However, in large memory-intensive applications or applications where there are a lot of simultaneous users such as a high-traffic web site, you might need to pay attention to this.
But this leads us to a best practice:

BEST PRACTICE: Don't attempt to help out the process of Garbage Collection by setting objects you are finished with equal to nothing.

Introducing Destructors

Before the object is destroyed, you can write code to clean up your application, which might include closing any open references to files or databases. This is called a Destructor in Object Oriented terminology, and to create a Destructor in Visual Basic, use the Finalize method in your class like so:

Visual Basic:

```
Protected Sub Finalize()

End Sub
```

How do you call this method from your application? You don't. If this method is present, the .NET Runtime will automatically execute the code in this method for you right before the object is being destroyed.

Why use a Destructor? In many classes you will create, you won't need to. You may choose to use a destructor if:

- Your class is responsible for opening files. If the class is to be destroyed, then you need to close the files it current has opened. Not doing so may corrupt the file, or prevent other parts of your application from accessing the file at a later time.
- Your class keeps a connection to a database open as long as an instance of the class is being used. This is not a good idea for many reasons, but I've seen it done this way and you could use a destructor for this purpose.
- You want to save the current state of the object to a database or a file for later use within your application.
- Your class is responsible for working directly with the Windows Application Programming Interface (API) and must carefully "disconnect" from Windows so that the application or Windows (or both) do not become disabled.

All of the above reasons are more advanced topics and you'll learn more about them as you continue to learn about programming in .NET.

Before I close, there is another, more advanced type of Destructor called Dispose. I won't cover this now, but it's a more deterministic way of destroying an object. By "deterministic" I mean that your application can be more proactive about destroying objects rather than just letting their reference go out of scope and allowing the Garbage Collector to find them. It's analogous to calling the garbage collector in your home town and requesting them to pickup that old refrigerator that's been in your garage for the last six months. Sure, they would find it as they routinely troll around the neighborhood on Monday, but you need to get rid of it NOW.

Object Oriented Programming Using C# 2005

Introduction

This chapter accompanies video 6, “Object Oriented Programming Fundamentals”.

Object Oriented Programming seeks to reduce the complexity of creating large applications by breaking the application down into smaller, manageable classes of code. Each class represents an idea, whether tangible and concrete (such as a Product or Employee) or conceptual (such as Inventory or Order).

In the past, most programmers wrote their programs in a coding style called Procedural Programming. As programmers used this style (or philosophy) of how to structure entire applications, its flaws become more glaring. One problem with procedural programming was the over-use of global variables (variables that could be used at any time anywhere in your application). Also, this style lent itself to coding practices that made the code hard to read and debug, and hard to maintain over a number of years. While procedural code is not inherently evil, carelessness caused problems as developers tried to fix bugs … changing a single line of code would have major devastating consequences in other parts of the application because the entire application was so interdependent. To make changes, programmers had to re-write entire applications because it was less work than trying to untangle the mess of code that they were left with.

Object Oriented Analysis, Design and Programming sought to improve on the entire software development process by taking a simplified approach to application design that more closely resembled everyday life.

You won't get very far in modern software development before hearing about Object Oriented Programming. It’s also known as “OO” for Object Oriented, and the purpose of OO programming is to reduce the complexity involved with writing and maintaining applications.

In real life, everything around us has properties and methods, although we generally don't think about life in these terms. For example, a Pencil has a color, a length, a lead quality ("bring your #2 pencils") ... and it also has methods, or things it can do like Write and Erase.

The basic idea behind object oriented programming is that if we can write code that mimics real world objects, like pencils, cars, customers, payments, etc. then it will make programming a lot easier. This is a gross over-simplification however it's a starting point.

Consider a car as an example of a real-world object oriented system that is comprised of many discrete parts. The car itself is an object. The engine inside the car is an object. The exhaust system is an object that contains even more objects. The engine depends on the exhaust system, but it has no idea how the exhaust system works. Even though the car parts don't know much about each other – except how to connect to each other – amazingly the parts all do their individual jobs and the end result is a working machine. If one part goes bad, it can be replaced without adversely affecting the rest of the car.

Additionally, you don't have to know anything about how a car works in order to drive it. You sit down in a comfortable seat, turn on the ignition key, and operate the steering wheel, the gas pedal, the brake and occasionally a turn signal. However, your instructions trigger hundreds of individual parts without you having to worry about them. This focuses on a major OO concept called Encapsulation. All of the powerful engine and navigation components are encapsulated behind two or three controls that you (the user) work with.

To extend this example further, there are specifications that define what a car is in general – if not in reality, then at least in our minds -- and each automaker inherits and extends those specifications to give their car a unique look, performance benefit or price benefit. In other words, each automaker inherits from the ideal "car" to create a more specialized car. Still, from a programmer's perspective, the Hyundai and the Mercedes have more in common than they have differences. The same general engine design, exhaust system design, number of tires, operation of the steering wheel, seats, pedals, turning signal, etc. exist in both cars. They both borrow from the design and mechanics of how a car works. In software terms, this is referred to as Inheritance, another important concept in Object Oriented Programming.

Also, all cars can go to the same gas stations and fill up using the same gas pumps, despite the fact that a Mercedes may cost $60,000 more than a Hyundai. This is known as Polymorphism, meaning that even though there are different objects, you can interact with them the same way because they both derive from the ideal car.

So, when defined in this way, I hope you can begin to see the value of looking at software the way we analyze the world around us.

For this lesson, we'll focus on the most basic building block of OO programming, and that is the Class. A class is simply a blueprint. The Class defines the fields, properties, methods and events that an object will have.

A class is a blueprint only; its only job is to define what data it should store (the variables, or rather Fields) and what actions the object should take (the procedures, or rather the Properties and Methods). But since a class is just a blueprint, it is not the ACTUAL object … just a definition of the object's appearance, behaviors, etc.

Just like in the real world, you take a blueprint to a contractor, or take the recipe to a baker and they create an instance of that blueprint, resulting in a house or a cookie respectively. In our case, we'll create an instance of the class. The instance is a manifestation of that class, which we then call an Object. So lets define a class:

C#:

```
class Car
{

}
```

This class does nothing. So let's add some fields, properties and methods.

C#:

```
public class Car
{
     private string make;
     private string model;

     public string Make
     {
         get { return make; }
         set { make = value; }
     }

     public string Drive()
     {
          if (make == “Oldsmobile”)
          {
               return “Chicago”;
          }
          else
          {
               return “Toledo”;
     }
}
```

Admittedly, this particular class still doesn’t do much but will allow us to see what a real class looks like. First, it has two private fields and a public function. We talked very briefly about scope in video lesson 4 and 5, and the words private and public have special meaning in terms of scope (visibility). I'll have more to say about it later in this chapter as we look at Properties, Fields and Methods.

Before we can interact with the Make property or the Drive function, we must create an instance of the Car class; or rather, we must instantiate the Car object.

C#:

```
Car myCar;
myCar = new Car();
string location;
myCar.Make = "Oldsmobile";
myLocation = myCar.Drive();
```

The most important part of this example is the first two lines of code. First of all we dimension a variable myCar that is of type Car. If you remember back to video lesson 4 we worked with simple value types like strings and integers. Now we are working with more complex reference types, AND we are creating our own custom types!

The second line of code is where the magic actually happens. In line 1 we created a variable of type Car, but as of yet it is not actually a Car. At the moment we use the new keyword, a Car object is created in memory and we store a reference to that new Car in the myCar variable. Let me clarify this. All the information about the newly created Car object is stored in your computer's memory. The variable myCar is given the address of that memory space, which is called a "reference". Now, whenever you use the word "myCar" in code, it just refers the compiler to the memory address that represents that instance of the Car object.

There's a subtle difference between dimensioning a variable as a type (or class) and actually creating an instance of that type (or class). This will become more apparent as we talk about Class Constructors a little bit later on.

Almost everything you work with in C#, whether you know it or not, is a Class. For example, look at the code examples from the previous lessons.

There is a lot more to learn, but in this lesson we've introduced Object Oriented Programming and have shown some fundamentals about classes.

Let's now look at what goes inside of Classes, namely Fields, Properties and Methods.

Fields

Let’s look at the Car class we created earlier in this chapter:

C#:

```
public class Car
{

     private string _make;
     private string _model;

     public string Make
     {
          get { return _make; }
          set { _make = value; }
     }

     public string Drive()
     {
          if (_make=="Oldsmobile")
          {
               return "Chicago";
          }
         else
          {
               return "Toledo";
          }
     }
}
```

First lines of code are fields. Fields are simply variables defined within a class, outside the scope of a method (a variable, but with scope that extends outside of a given method). They can be private like we have in the Car example, or they can be public.

Ideally, you should never use Public Fields, but rather use Public Properties instead. We'll talk about that more in a moment, but let’s add three more BEST PRACTICEs to our ongoing list:

BEST PRACTICE: Instead of public fields, use public properties and use private fields as the container for the value and properties as mutators and accessors to the hidden value.

BEST PRACTICE: Use an underscore as an prefix to your private field, then use camel casing for the rest of the name. Example:

_engineSize or _elapsedMileage

BEST PRACTICE: use camel case when naming private methods. Use pascal case when naming public methods and properties.

Camel Case == engineSize or elapsedMileage
Pascal Case == EngineSize or ElapsedMileage

Properties

Lets add a public field for a moment just to show how we would reference that class variable in our code:

(This is just a re-cap of the Car class … all we're doing is adding a Public field)

C#:

```
public class Car
{

     public int ElapsedMileage;

}
```

(This code would be used to create an instance of the Car object and use the new Public field we just added)

C#:

```
Car myCar = new Car();

if (myCar.ElapsedMileage == 0)
{
     myCar.ElapsedMileage = 10000000;
}
```

This illustrates two ideas. First, it shows how to use the . (dot) syntax to access public fields and set or retrieve their values. Second, it shows the problem with public fields. You could set their values to nonsensical values. Very few cars have traveled 10 million miles. We would obviously like to limit that to a sane amount. We can do validation (also known as "sanity checks") on values, or filter the value of a field by using a property instead. Lets change the ElapsedMileage from a public field to a private field:

C#:

```
private int _elapsedMileage;
```

Notice that I changed the underscore and capitalization of _elapsedMileage in keeping with our Best Practices (defined above). Private member variables should be camel cased and be prefixed with an underscore.
And then create a public property called ElapsedMileage:

C#:

```
public int ElapsedMileage
{
     get
     {
          return 5;
     }
     set
     {
          // Do nothing
     }
}
```

Notice that there are two distinct parts of the property statement ... a get and a set. The get retrieves and returns a value to the code that calls it. The set assigns a value (more about this in a moment). Note that the first line of the property statement does nothing by itself; we have to decide which value to return when the get statement is called and what to do with the value that is passed into the parameter of the set statement. We could ignore these values, "hard code" them, or whatever we like. Typically, we'll do this:

C#:

```
public int ElapsedMileage
{
     get
     {
          // You could do some validation
          // or modification here
          return _elapsedMileage;
     }
     set
     {
          // You could do some validation
          // or modification here
          if (value > 1000000)
          {
               _elapsedValue = 1000000;
          }
          else
          {
               _elapsedValue = value;
          }
     }
}
```

In this example, we chose to use a private field elapsedMileage to store the parameter of the set statement. However, we do a little due diligence to make sure that someone didn't enter a bogus amount of miles. Then we use the elapsedMileage field again when someone calls the get statement.

So now lets see how these are used in code:

C#:

```
Car myCar = new Car();
if (myCar.ElapsedMileage == 0)
{
     myCar.ElapsedMileage = 10000000;
}
```

Yes, this code is correct. It is identical to the code we used when explaining public fields. You may wonder WHERE the call to the get and set statements is located. Actually, the calls are implied by the fact that we are reading the value in the 'if' statement (implied a call to the get method), and the fact that we are assigning it a value in the myCar.ElapsedMileage = 10000000 statement (implies a call to the set method).
If we add one more statement:

C#:

```
MessageBox.Show(myCar.ElapsedMileage.ToString());
```

… what value do you think will be displayed? If you said 1 million you'd be correct. That is because of the way that we wrote our set statement to filter out insane numbers like 10 million.
One final point of clarification about the set; consider this line of code that was in the set statement above:

C#:

```
_elapsedValue = value;
```

Where did the "value" identifier come from? In this case, C# uses this token ("value") to represent the data that the property is being assigned to. It's similar to an input parameter, but it's built in and you can't change the name of the identifier to anything else (like "mySetValue", for example).

Methods

We’ve already discussed methods earlier in this lesson, and stated that you must enclose a method in a class, but at that point it probably didn’t mean a lot. Let’s continue our discussion of Methods by considering the method used in our Car class example:

C#:

```
public string Drive()
{
     if (make=="Oldsmobile")
     {
          return "Chicago";
     }
     else
     {
          return "Toledo";
     }
}
```

Methods can have either Private or Public visibility. (Actually there are a few other types of visibility, but let’s just focus on these two for the time being.) Many times, you'll want Private methods to be used as “helper” functions or utilities that hide how the class does its work internally. Ideally, classes are like mysterious "black boxes" to the code that calls them. You know nothing about how the object works except for the Public properties and methods, and that is good. This is known as "Encapsulation" in Object Oriented programming terminology. So, suppose that we were to change the Drive method from a public method to a private one. Then suppose we were to attempt to call the private Drive method from our Form1_Load event … what do you think would happen? We would get an error that says:

‘Car.Drive()’ is inaccessible due to its protection level
In other words, because the method is private, it is protected from other classes attempting to call that particular method.

Overloading Methods

Consider the following two Drive methods closely:

C#:

```
public string Drive()
{
     if (make=="Oldsmobile")
     {
          return "Chicago";
     }
     else
     {
          return "Toledo";
     }
}

public string Drive(string _make)
{
     if (_make=="Oldsmobile")
     {
          return "Chicago";
     }
     else
     {
          return "Toledo";
     }
}
```

This is called Overloading the Method. We've overloaded it with two different ways of calling the function. We can either call it passing in no parameters, or call it passing in the _make. Each method returns a string, but the parameters and how the method works internally can be different. In order to overload a method, the method signature must be different. "Method Signature" is just a technical term that means for the unique set of input parameters in a method. Why would you overload a method? When you create classes, you don't always know how they might be used up front in your application. Sometimes you don't have all the data readily available to you that you would need to call a method, so you need options on how to call the method. Also, overloaded methods with additional parameters can offer and extended set of functionality, much like the MessageBox.Show() method which allows you to just pass a message to be displayed, or other overloaded versions which allow you to pass in the title for the message box, icons, which buttons are available, and much more.

Object Lifetime

Like many living objects in the real world, code objects have a birth, a life and a death. You can write code that executes when an object is born and when it is about to die.

We've talked about instantiating an object from its class definition, which means we take the Class and create an object from it. But we haven't told the entire story. Let's look at what happens when we create an instance of a class, and we'll talk about what happens after we finish using it.

Introducing Constructors

First of all, classes have Constructors, which is a special method that allows you to initialize your fields, or whatever else you may want to do when your object is first created. In C#, you create a Constructor using the same name as the class in the form of a method. For example, this is what the constructor for the Car class would look like:

C#:

```
public class Car
{

     public Car()
     {
          make = "Unknown";
          model  = "Unknown";
          elapsedMileage =0;
     }

}
```

Here we've set our private fields to default values. The Constructor gets called whenever the new keyword is used in association with our class. Here's an example of code you might see in the Form1_Load method, or some method that would need to create an instance of the Car class:

C#:

```
myCar = new Car();
```

Also, the constructor is ALWAYS the first code to execute within your class, however having a Constructor in your class is optional. While Constructors are not required, it is a good idea to use constructors to initialize the private property (private member) values of your new object.

I've used the term "initialize" several times; let me explain what I mean. Initialization is the process of taking steps to ensure that your object will function properly by the time the object is used in an application. That means different things based on how you design your object. For example, you may choose to create a connection to a database, or create instances of child objects, or set variables (like private fields) to default values, etc. The constructor is the perfect place to execute code that MUST RUN in order for your object to perform correctly.

BEST PRACTICE: Use Constructors to initialize values for a newly created object.

Overloading Constructors

Since a Constructor is simply a method, you can overload the Constructor. Recall from earlier in this chapter that overloading a method means that you can have different implementations of your method based on a different method signature.

I could create another Constructor that allows me to initialize the values as soon as I create an instance of the class:

C#:

```
public Car(string make, string model, int elapsedMileage)
{
     make = _make;
     model = _model;
     elapsedMileage = _elapsedMileage;
}
```

Then, this is how I would use that constructor when I create a new instance of the class:

C#:

```
myCar = new Car("Nissan", "Altima", 31000);
```

Death of an Object

So we understand what happens when we create an object, but what happens when we are finished using an object? This isn't an easy answer, so try to understand what I'm about to say involves some concepts that are rather advanced and I'm not telling you everything.

We sometimes talk about things being in a plastic bubble, maybe you've seen the movie "The Boy in the Plastic Bubble" or watched the Sienfeld episode with the "Bubble Boy". The bubble protects people with weakened immune systems from the outside world. The Bubble is a totally controlled environment. In a like manner, that is what the .NET Runtime does for your applications. It protects your application from the outside world, so that other programs can't corrupt its memory space, and it cleans up after your program when it's finished. Protecting an application's allocated memory prevents memory leaks or other bugs from shutting down your application immediately. To guard against this, the .NET Runtime has a Garbage Collector, which is a process within the .NET Runtime that searches for object references in your code that are no longer needed. When it finds an unused object it disposes of it, or rather, it removes the object from memory.

As your code is executing, the .NET Runtime will flag those objects that it knows it won't need again, and after a while the Garbage Collector comes through and disposes of all the objects that are flagged for destruction. Right before the Garbage Collector destroys your object, you will get a last second chance to execute some code, which is called a Destructor … I'll cover that in the next section.

Many developers try to help the .NET Runtime by doing the following:

C#:

```
myCar = null;
```

However this is not necessary. When a given procedure finishes processing, the variables that were defined in that procedure go "out of scope" which means that their values are no longer available -- the task is complete and the variables are no longer needed. As the .NET Runtime executes, it looks ahead and decides when it needs to destroy an object, based on whether or not the object's reference is used in future code.
So, by waiting until the very end of the procedure to add this code:

C#:

```
myCar = null;
```

… you might actually be hindering rather than helping the .NET Runtime to clean up memory sooner. Although, lets be honest, it really won't make that big of a difference in smaller programs. However, in large memory-intensive applications or applications where there are a lot of simultaneous users such as a high-traffic web site, you might need to pay attention to this.

But this leads us to a best practice:

BEST PRACTICE: Don't attempt to help out the process of Garbage Collection by setting objects you are finished with equal to null.

Introducing Destructors

Before the object is destroyed, you can write code to clean up your application, which might include closing any open references to files or databases. This is called a Destructor in Object Oriented terminology, and to create a Destructor in C#, create a method that is named the same as the class preceded by the tilde character (~):

C#:

```
~Car()
{

}
```

How do you call this method from your application? You don't. If this method is present, the .NET Runtime will automatically execute the code in this method for you right before the object is being destroyed.

Why use a Destructor? In many classes you will create, you won't need to. You may choose to use a destructor if:

- Your class is responsible for opening files. If the class is to be destroyed, then you need to close the files it current has opened. Not doing so may corrupt the file, or prevent other parts of your application from accessing the file at a later time.
- Your class keeps a connection to a database open as long as an instance of the class is being used. This is not a good idea for many reasons, but I've seen it done this way and you could use a destructor for this purpose
- You want to save the current state of the object to a database or a file for later use within your application.
- Your class is responsible for working directly with the Windows Application Programming Interface (API) and must carefully "disconnect" from Windows so that the application or Windows (or both) do not become disabled.

All of the above reasons are more advanced topics and you'll learn more about them as you continue to learn about programming in .NET.

What is the .NET Framework?

Introduction

This chapter is an overview of the purpose and the parts of the .NET Framework. It is provided as a supplement to video lesson 7, “Getting to know the .NET Framework”.

Like many geeks, my favorite movie is the Matrix. As the movie starts, you can't quite understand what this mysterious term means to Neo and the others who are trapped inside of it. Then Morpheous explains what the Matrix is, and while you don't understand it all, your mind just expanded to understand the events that have been happening to Neo. To loosely quote Morpheous, who in turn references Alice in Wonderland: "we're about to fall deeper into the rabbit hole."

What is .NET?

.NET by itself is simply a marketing term that describes many different software products (and services) from Microsoft. The idea is that these products all work seamlessly together and allow businesses to create eCommerce web sites and allow partners to exchange data almost effortlessly. However, this caused a lot of confusion among Microsoft’s customers, so Microsoft is working on renaming their server products (Windows 2003, Biztalk, Commerce Server, etc.) to more clearly define what .NET is. From a software development perspective, the heart of .NET is the .NET Framework, which is a collection of tools to allow developers to create applications.

Tools include:

.NET Runtime - this is the core component. It is also known as the “Common Language Runtime” or just “CLR”. The runtime is a type of software known as a “virtual machine” that acts as a mediator between the Windows operating system and the code the programmers write. It protects end-users from malicious code and protects developers from managing the intricacies of dealing with memory (like we learned the Garbage Collector does in the previous lesson), the file system, and other lower level functionality. Also, a huge benefit is that it doesn’t require that the programmer write different code for each version of Windows. The Runtime handles those complexities and allows software developers to focus on the functionality of their application instead. In theory, the Runtime could be ported to other operating systems such as Linux so that an application you create for Windows will run on Macs or Linux machines. This is what Java attempted to do with their virtual machine, which .NET was heavily influenced by. When your programs run, they run "inside" or are controlled by the .NET Runtime. When an end-user clicks an icon to open your program on their computer, the first thing your program will automatically do is "ask" the .NET Runtime to "host" it. Therefore, any computer you want to use your application on must have the .NET Framework installed or else it will not work on that computer. We do not cover how to distribute your application in the BEGIN01 series, however there are other videos on LearnVisualStudio.NET that cover this topic (see video "3550 - Introduction to Deploying .NET Apps" for more information.)

.NET Framework Class Library – also known as the "FCL". This is what you will become intimately familiar with. It contains hundreds of classes, each with dozens of properties and methods that allow you to do almost anything you would want to within your program. You won't get very far without using the Class Library -- database access, sending emails, accessing files on a hard drive, creating dynamic web pages -- everything you want to do is made possible through this set of code that you reference within your applications.

.NET Language Compilers - These are a set of programs that take the source code you write in a particular language (such as Visual Basic.NET, C#, J# or managed C++), and compile them into the Microsoft Intermediate Language (MSIL, or just IL) which is then run by the .NET Runtime. The compiler for Visual Basic.NET is vb.exe. Each programming language has its own compiler that knows how to interpret source code you write into MSIL and store that MSIL in an Assembly file. The compiler also alerts you to problems compilation errors in your code due to mistakes in typing, incorrect logic, etc.

Other utilities - There are over a dozen other tools that come with the .NET Framework for various purposes. Many of these you may never need, and some are used behind the scenes by Visual Studio.NET.

There are MANY other features of .NET, such as a common type system that allows cross-language interoperability, the inner workings of Assemblies (what your application gets compiled into), application domains for isolation of applications and their processes, and many complex topics. If you want to really dig in deep and understand the .NET Framework, you must purchase and read this book at least twice:

This is the VB version:

Applied Microsoft .NET Framework Programming in Visual Basic .NET

By Jeffery Richter and Francesco Balena
Published by Microsoft Press
ISBN: 0735617872
http://www.bookpool.com/.x/symeao4zer/sm/0735617872

And this is the C# version:

Applied Microsoft .NET Framework Programming

By Jeffery Richter
Published by Microsoft Press
ISBN: 0735614229
http://www.bookpool.com/.x/io36m6jzo4/sm/0735614229

Caution: it is NOT a book designed for beginners.

Namespaces in the .NET Framework Class Library

The .NET Framework Class library has hundreds, if not thousands, of classes. In order to find the classes you are looking for, and as a result of good Object Oriented design, the classes are divided up into Namespaces. A namespace is a way of segregating classes into common functionality and also allows two classes with the same name to be unique. For example, I'm not the only Robert in Texas, but I'm one of only a couple of Robert Tabors in Texas. Furthermore, I'm the only Robert Theron Tabor in Texas, and probably the world, so by adding names that makes my otherwise common name unique. In a like manner, it would be very difficult for 3000+ classes to have different names, especially when some of them have similar functionality that is used at different times or for slightly different purposes.

There are dozens of important namespaces ... here are a few that you'll get to know over the next few video lessons:

System
System.Data
System.Data.SqlClient
System.IO
System.Exception
System.Web

While the system Namespace is one of the most important, it's not the only namespace root. If we wanted to create a connection to a database, we would have to write the following code:

Visual Basic:

```
Dim con As System.Data.SqlClient.SqlConnection
```

C#:

```
System.Data.SqlClient.SqlConnection con;
```

Wow, that is a long name! Furthermore, if I wanted to create a command object and a data reader from that same namespace, I'd have to add the next two lines of code:

Visual Basic:

```
Dim cmd As System.Data.SqlClient.SqlCommand
Dim dr As System.Data.SqlClient.SqlDataReader
```

C#:

```
System.Data.SqlClient.SqlCommand cmd;
System.Data.SqlClient.SqlDataReader dr;
```

To reduce the amount of carpel tunnel, Microsoft allows us to use an Imports statement which makes all the classes in a given namespace available without having to type its full name each time.

Visual Basic:

```
Imports System.Data.SqlClient
...

Dim con As SqlConnection
Dim cmd As SqlCommand
Dim dr As SqlDataReader
```

C#:

```
using System.Data.SqlClient;
...

SqlConnection con;
SqlCommand cmd;
SqlDataReader dr;
```

Namespaces in your Applications

When you create an applications using Visual Studio.NET, a default namespace will be automatically created. Suppose you create a new project and you name it “MyProgram”. The root namespace for your application is called “MyProgram” and if you add a class to your application called “MyClass”, the fully-qualified name of the class would be “MyProgram.MyClass”. Why do you need namespaces in your applications? This prevents a naming conflict between the names of your classes and the names of classes in the .NET Framework Class Library. Suppose you create a class in your application called Format. You may not realize it, but there is a class called Format in the .NET Framework class library, too! Without providing a “middle name and a surname” (just like people have to distinguish them from one another) for our class, the compiler would have no idea which class to create an instance of. That is why your Format class is automatically assigned the name “MyProgram.Format” which differentiates it from the “System.Windows.Forms.DataFormats.Format” class.

Also, if you or your company are responsible for creating a re-usable class library (a .DLL file, or rather, Assembly that can be shared across one or more applications), a namespace will prevent a naming conflict between your class library and one created by someone else that is used within the same application.

You can change the root namespace for your project in the Project Properties dialog in the Common Properties tab.
You also can define as many namespaces in code as you want to using the following structure:

Visual Basic:

```
Namespace YourNamespaceName

. . .

End Namespace
```

C#:

```
namespace YourNamespaceName
{

. . .

}
```

However, your root namespace will always be at the beginning. Consider this case where your project name is MyProject and you have a class file with the following code:

Visual Basic:

```
Namespace MyNamespace

     Public Class MyClass
          . . .
     End Class

End Namespace
```

C#:

```
namespace MyNamespace {
      public class MyClass {
            …
      }
}
```

The fully qualified name for this class would then be:

```
MyProgram.MyNamespace.MyClass
```

Confused? Don't worry. While this information will ultimately affect you as you build larger and larger applications, it won't get in the way of creating usable applications today. I wanted to introduce this thought to you because it will pop up everywhere and is central to developing .NET applications.

Creating and Obtaining Data from a Database

Introduction

This chapter is to be read with video lessons 8 and 9, “Obtaining Data from a SQL Server 2005 Express Edition Database” and “Databinding Data to User Interface Controls”, respectively.

Databases are files or systems that organize data for easy storage and retrieval. Working with Databases will probably become the single most important use for your coding skills if you plan to work for a typical IT department as a programmer.

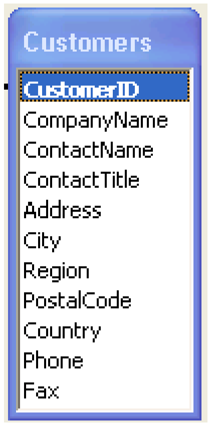
There are many different types of databases from different vendors. Each database shares a common structure and terminology. This is known as a Relational Database. You will need to use a sub-set of the .NET Framework called ADO.NET, which stands for ActiveX Data Objects. Don’t worry … there’s nothing about the history of ADO that is pertinent right now. What IS important is that it contains a series of classes that allow you to interact with all types of different databases very easily.

Understanding Databases

Most of the applications I write involve accessing information in a database, whether I just select records to display on screen, or insert, update or delete records.

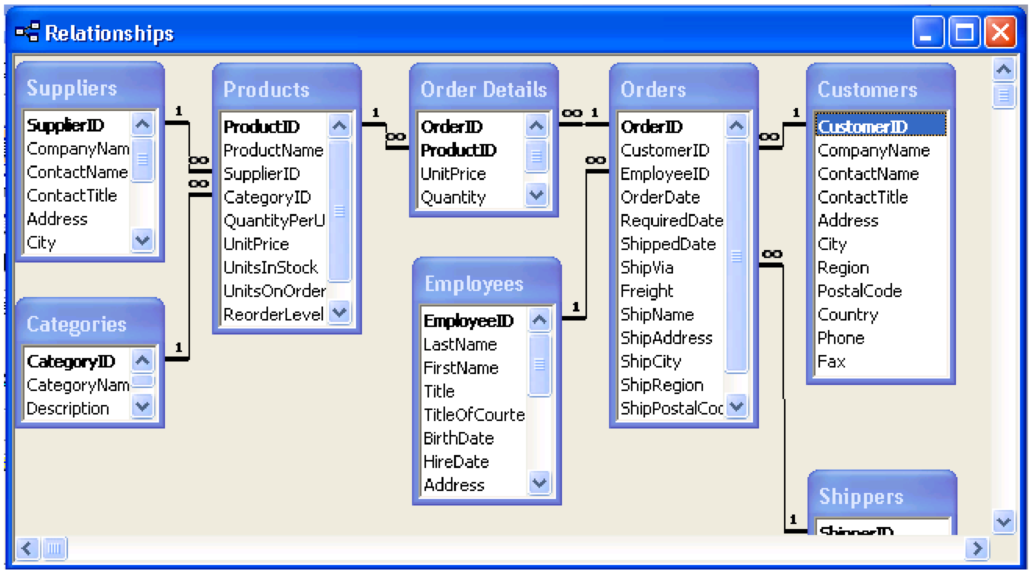
A database is a file that has data structured in such a way that the data is organized and easily searched for, or modified. A database can be created by Microsoft Access for simple applications … or better yet, you can take advantage of Sql Server 2005 Express Edition which is installed with Visual C# 2005 Express Edition and Visual Basic 2005 Express Edition and is integrated directly into the IDE via the Database Explorer window. Or in larger companies or for high-volume web site, you may want to use a Database Management System like SQL Server 2000 or 20005 or Oracle 9i Database. These larger systems don't allow you to access the data file yourself, but have a program (called a Database Server) that accepts your request, performs the request and delivers the results (if any). The benefit is that it can accommodate more computers that try to access the data at the same time (i.e., better performance for multiple concurrent users).

Regardless of the database, data is organized into Tables. Tables have columns, which are properties of the table. So, for example, there is a database called Northwind that ships with Microsoft Access as an example. One of the tables looks like this:

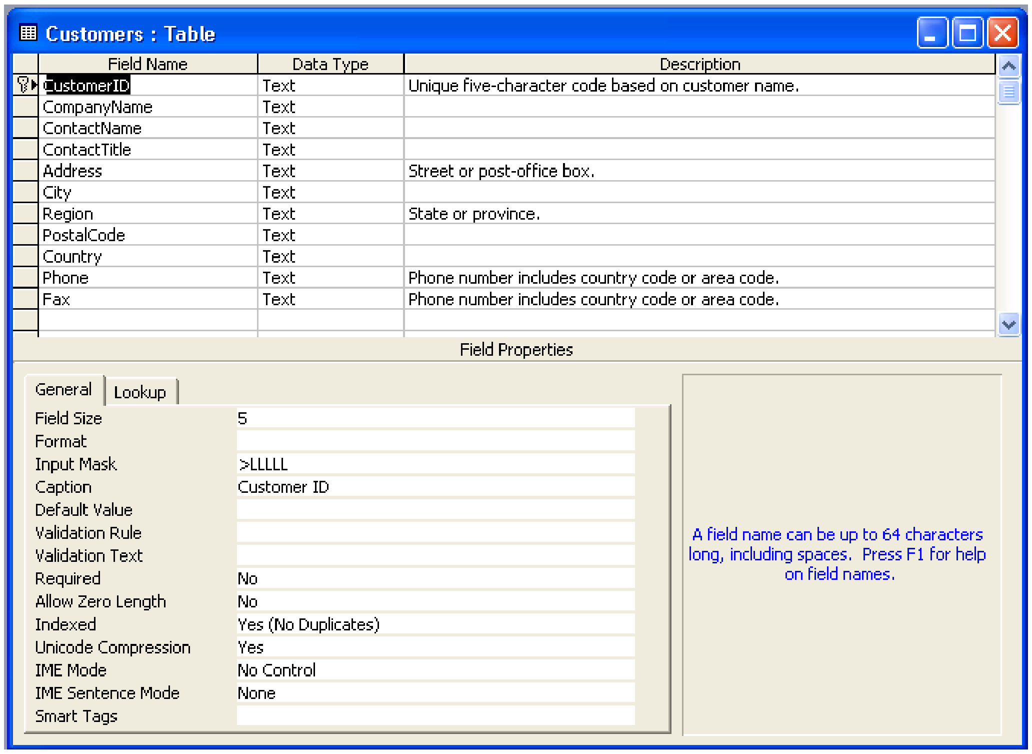


Note: while we are looking at a Microsoft Access database, these similar styles of views are available from within the Express Edition tools for databases created and managed by Sql Server 2005 Express Edition, as well as other database management tools from Microsoft.

We are looking at a common way that the structure of a table is represented – in the form of a picture. Here's a picture of ALL the tables in the Northwind database:



The lines that connect each of the tables are called Relationships. We'll talk about them more in a bit. There's another way to view the structure of a table in a database:



Customers : Table

Field Name	Data Type	Description
CustomerID	Text	Unique five-character code based on customer name.
CompanyName	Text	
ContactName	Text	
ContactTitle	Text	
Address	Text	Street or post-office box.
City	Text	
Region	Text	State or province.
PostalCode	Text	
Country	Text	
Phone	Text	Phone number includes country code or area code.
Fax	Text	Phone number includes country code or area code.

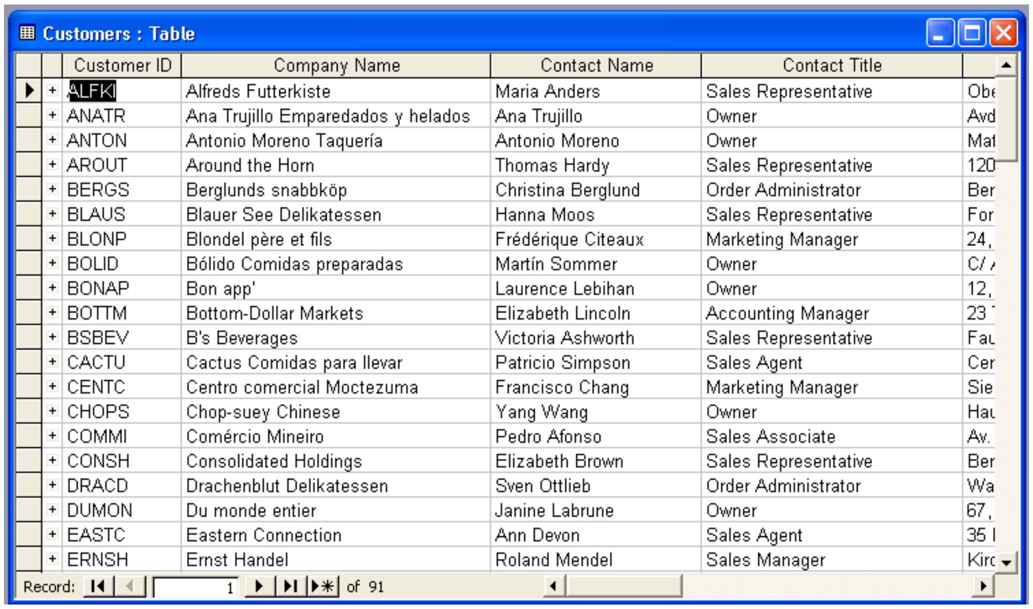
Field Properties

Field Size	5
Format	
Input Mask	>LLLLL
Caption	Customer ID
Default Value	
Validation Rule	
Validation Text	
Required	No
Allow Zero Length	No
Indexed	Yes (No Duplicates)
Unicode Compression	Yes
IME Mode	No Control
IME Sentence Mode	None
Smart Tags	

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

This is called a Design View. A table has columns and rows. It might be easy at this point to think of a table as a spreadsheet, like the type you use in Microsoft Excel. The columns are the structure of the table. The columns define WHAT type of information will be saved in that table. In this case, the Customers table has a structure of 11 columns (CustomerID, CompanyName … Fax) A single customer in the Northwind database will have one row of data in this table. A row represents a collection of fields of information that are related together conceptually. The Design View (above) allows you as the database designer to add or remove columns from a table.

If you want to add or remove rows from a table, you'll need yet ANOTHER different view:



Customers : Table

Customer ID	Company Name	Contact Name	Contact Title	
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obe
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avd
ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Mat
AROUT	Around the Horn	Thomas Hardy	Sales Representative	120
BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Ber
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	For
BLONP	Blondel père et fils	Frédérique Citeaux	Marketing Manager	24,
BOLID	Bólido Comidas preparadas	Martín Sommer	Owner	C/
BONAP	Bon app'	Laurence Lebihan	Owner	12,
BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	23
BSBEV	B's Beverages	Victoria Ashworth	Sales Representative	Fau
CACTU	Cactus Comidas para llevar	Patricio Simpson	Sales Agent	Cer
CENTC	Centro comercial Moctezuma	Francisco Chang	Marketing Manager	Sie
CHOPS	Chop-suey Chinese	Yang Wang	Owner	Hau
COMMI	Comércio Mineiro	Pedro Afonso	Sales Associate	Av.
CONSH	Consolidated Holdings	Elizabeth Brown	Sales Representative	Ber
DRACD	Drachenblut Delikatessen	Sven Ottlieb	Order Administrator	Wa
DUMON	Du monde entier	Janine Labrune	Owner	67,
EASTC	Eastern Connection	Ann Devon	Sales Agent	35
ERNSH	Ernst Handel	Roland Mendel	Sales Manager	Kirc

Record: 1 of 91

This is known as the Datasheet View. Look at the very top of this view … each   column has a header in light gray. That represents the name of each column. This will coincide with each column in the database table.

The each record is represented in the rows below the first row (the header row). Each row has a set of data values that correspond to each column in the table. Again, this looks a lot like an Excel spreadsheet. We can add a new row by typing into the last row of the table, and moving to the next line. OR we can create a program that allows a user (or many users) to add new records into our database without having to use Microsoft Access. The reason we would want to do that is to provide a more user-friendly interface to the end user, or to prevent the user from modifying the data in a way that violates our business' practices or policies. These are called "Business Rules" … we use that term often, since that is the reason why many of us are gainfully employed as programmers … to create programs that enforce business rules.

Each column can have several properties associated with it. For example, each column will have a name, a data type (just like variables have), and depending on the data type, a size.

Additionally, columns can be used to AutoNumber so they make good columns for identifiers (or IDs … special columns used in primary/foreign key relationships.)

Relationships

In a Relational Database, there might be many tables, each one linked to other tables by a relationship. A relationship means that for one row in one table, there might be one or more rows related to it in another table. For example, one table might contain companies, and another table called employees. One column in the Companies table is CompanyID, which would be the Primary Key. In this case, the CompanyID is just a number that uniquely identifies this row in the table ... there can't be two Companies with the same CompanyID. Each row in the Employee table will have a corresponding CompanyID field, which makes it related to the Company table. This CompanyID field is called the Foreign Key.

SQL Statements

You can retrieve values from the database using SQL statements, which stands for Structured Query Language. SQL is like a programming language for databases. For example, this statement would retrieve all the rows from the Company table where their headquarters were based in Texas:

```
SELECT * FROM Customers WHERE State = 'TX'
```

And here is a statement that might update one Company's city, state and zipcode:

```
UPDATE Customers SET City = 'Burbank', State =
'IL' and Zipcode = 60459
```

While this might seem like a lot of information, it’s actually just the tip of the ice burg. We may want to create our own database to store information collected from a Guest Form on a web site, or to store our company's products and pricing information, or any thing else we can imagine.

ADO.NET

The .NET Framework allows us to work with all types of different databases, including Access, SQL Server 2005 Express Edition, Sql Server 2000, Sql Server 2005, Oracle and others through the user of a special set of .NET Framework classes called ADO.NET. ADO.NET provides two ways to work with data in a database.

Connected Database Access - In this first strategy, a connection to the database is opened and all the database operations such as selecting records, modifying records, deleting records, etc. is performed. Once you are finished making your changes, you can close the connection to the database.

Disconnected - In this strategy, a connection to the database is opened just long enough to get a set of records. Once the records are retrieved, the connection is closed. Then your code modifies the records, inserts new ones or deletes records and ADO.NET keeps track of all the changes. Then you command ADO.NET to connect back to the database and make those changes permanent by updating the actual database. The benefit of working with disconnected data is that you can free up system resources, such as database connections (in the case of SQL Server or Oracle) or reduce the amount of time the database is locked (when using Access).

In order to accomplish both of these approaches to data access, there are a series of classes in the System.Data namespace. I'll overview them at a high level. For more information, use the Help system or view additional videos on LearnVisualStudio.NET.

Connection object – This object manages the "handshake" between your application and the external data source (database).

Command object – This object manages the request your application will make to retrieve or change data in the data source. Typically this will take the form of a SQL statement. It also manages any parameters that are sent to the data source in addition to the SQL statement. A command object must be associated with a connection, and then executed in order for it to perform its duty.

DataReader object – This object is used in a connected data scenario. Once the command object sends a request to retrieve data from the data source, the data has to be retrieved into either a DataReader or a dataset (described below). Think of the DataReader as a straw that you use to deliver your Slurpee from the cup to your mouth. The DataReader retrieves the results from the query and allows your application to look at each individual row of data. Once your application is ready, you use the Read method of the DataReader to retrieve the next row of data until all rows have been retrieved. At that point, the connection to the database can be closed.

DataSet object – The dataset is a container for disconnected data. The dataset can hold data in multiple data tables, each data table representing one or more tables from one or more data sources, such as databases, XML files, text files and more. So, one dataset could contain a table from a SQL Server 2005 Express Edition database and an Access database. The data tables and their relations are defined in an XML Schema document called an XSD. Or, developers can define the structure of the data tables (including their columns names and data types, etc.) programmatically.

To extend the Slurpee analogy from the previous section about the DataReader … a DataReader is a straw. A DataSet is another cup that you transfer the Slurpee into. Once your application is finished with the Slurpee, you transfer it back into the original cup. So as you can see, the scope of the DataSet is much larger than that of the DataReader.

Changes to the dataset are recorded in a series of diffgrams, which essentially is a list of changes that your program makes to the dataset – data rows added, edited and deleted from the dataset. These diffgrams are kept until the changes are resolved back into the original data source through the use of the Update method of the associated DataAdapter (more about that object below).

The dataset and its contents are stored in memory for as long as the program keeps a reference to the dataset's instance. However, it has a unique ability to be persisted (saved) to XML and that XML can be stored into a file or sent over the network through the use of a Web Service. While it might not be apparent why you would want to do that right now, eventually you may face a situation where you need to store (or send the data to another computer for processing) for a prolonged period of time … even when the application that originally created the dataset is not running. This is a bit advanced, however it provides a great deal of utility for your application design.

DataAdapter object – As briefly noted in the paragraphs about the dataset, the DataAdapter contains a series of command objects. Each command has a particular function … a command object to perform a selection of the data from the original data source (so that the data can be stored for processing in the dataset), and command objects to add, edit and delete rows back to the original data source. It performs these operations using the diffgrams that were mentioned when discussing the dataset. Think of the DataAdapter as the conduit between the original data source (such as a database, file, etc.) and the in-memory representation of the data for use in your application (the dataset).

There are many others, however this should get you started.

By the way, with the exception of the DataSet, specific versions of the Connection, Command, DataReader and DataAdapters are provided for each specific type of data source you choose to use. So, for example, there is the SqlConnection, the SqlCommand, the SqlDataReader and the SqlDataAdapter, which are part of the System.Data.SqlClient namespace. Why do this? To optimize these objects and specialize them for the peculiarities that make each data source different.

ADO.NET 2.0

In ADO.NET 2.0, the original data objects (described above) do not go away, however much of their functionality has been encapsulated and made more easily usable, especially in those scenarios when you want to bind the data that has been retrieved from the original data source.

Here are a couple of those objects that are highlighted in video 9.

TableAdapter object – Combines the Connection, Command and DataAdapter functions into one easy to configure and use object. Its purpose is to retrieve data and put it in a dataset.

TableAdapters are available for specific types of data sources, such as databases (like SQL Server 2005 Express Edition), ODBC data sources, XML files, custom business objects and more.

BindingSource object – Manages the relationship between Windows user interface controls on your form to the associated dataset's datatable's columns as the user/application navigates from row to row.

There's also a new object that is available when creating web applications in ASP.NET 2.0

DataSource object – Similar to the table adapter, this ASP.NET 2.0 scriptable object makes it easy to connect to, select, modify and bind to data without writing any C# or Visual Basic code. This can simplify the integration of external data into your web sites for those who are more familiar with scripting (using an HTML-like syntax) rather than programming.

See the Visual Web Developer 2005 Express Edition for Beginners series (video number 8) for more details.

http://lab.msdn.microsoft.com/express/vwd/videos/

The DataSource also comes in a variety of flavors based on the underlying data source that you will be using, including Sql Server, Access, a custom business object, XML and more.

As a beginner, you might find that you can get very quick results by using the TableAdapter or DataSource object. However it's important to understand that they are built on top of other objects and learning about those as well can help you develop more robust applications.