# CS608 Lecture Notes

## Visual Basic.NET Programming

### Object-Oriented Programming

### Inheritance

## (Part I of II)

### (Lecture Notes 3A)

Prof. Abel Angel Rodriguez

**CHAPTER 8 INHERITANCE** ..... 3

**8.1 Introduction to Inheritance** ..... 3
8.1.1 Introduction to Inheritance ..... 3
8.1.2 Implementing Basic Inheritance ..... 5
8.1.3 Available Access to Base Class Members from SubClasses ..... 15

**8.2 Inheritance Concepts** ..... 16
8.2.1 Inheritance Features ..... 16
8.2.2 Method Overloading in Inheritance ..... 17
General Method Overloading Review ..... 17
Implementing Method Overloading in Inheritance ..... 18
8.2.3 Method Overriding ..... 25
Introduction ..... 25
Implementing Method Overriding ..... 25

# Chapter 8 Inheritance

## 8.1 Introduction to Inheritance

### 8.1.1 Introduction to Inheritance

**Reusability**

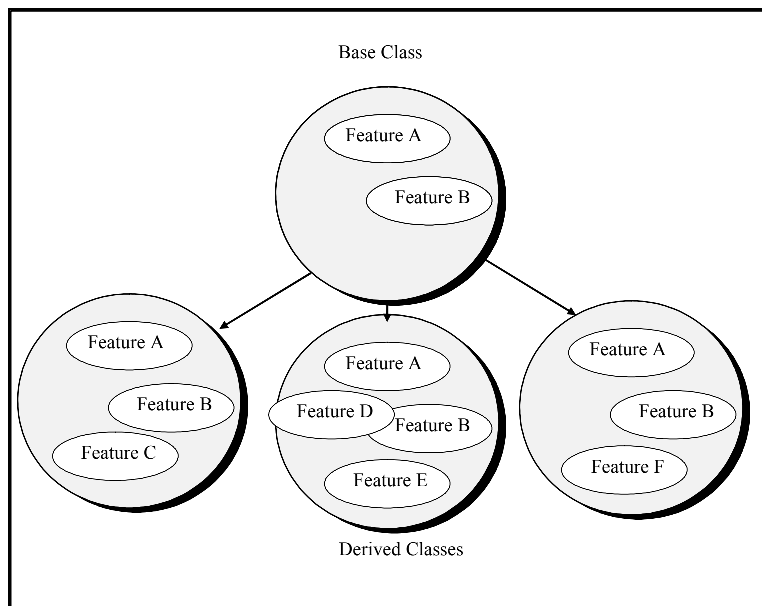
- Previously we introduced the concept of reusability. That is re-using objects that we create in one program into another.
- This concept has revolutionized the field of programming. Applications which took longer to developed are now being created at a much faster rate since objects from other applications are being reused, thus saving time on programming and testing.
- The Objects re-used have already been tested in previous programs so they are guaranteed to work safely thus yielding a robust program.
- This concept of *reusability* spawned a new software industry where companies were established whose sole business is to create ready tested Objects to sell to other software development houses.
- The main Object-Oriented Programming concept provided to implement reusability is ***Inheritance***.

**Inheritance**

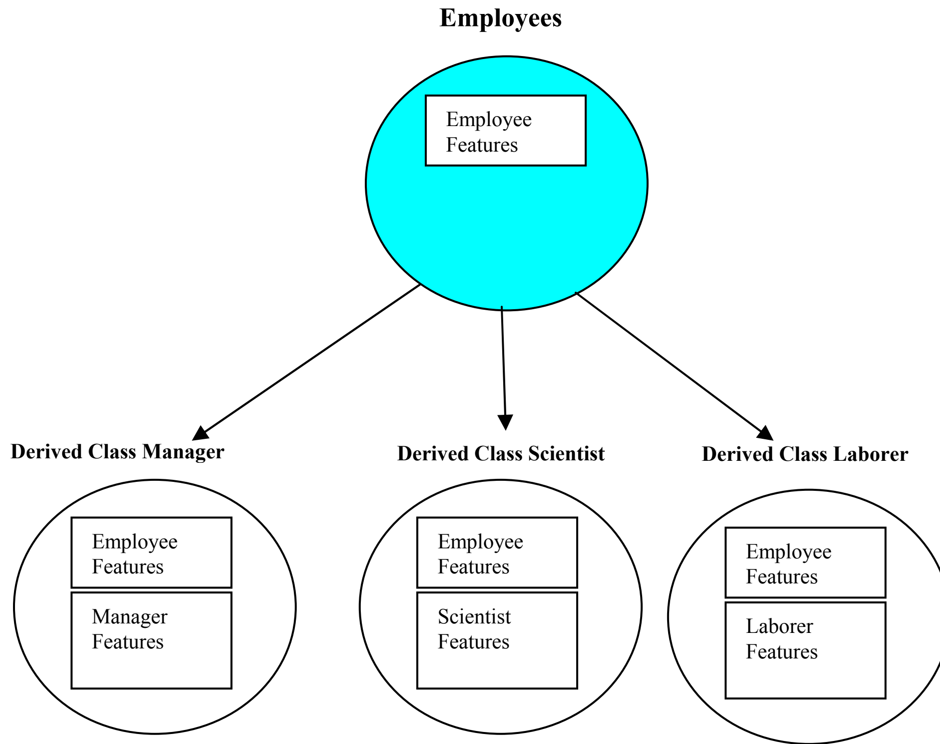
- Inheritance is probably the most powerful feature of Object-oriented programming.
- Inheritance is the process of creating new class, called ***Sub Class***, (*Derived Class*) from an existing parent class. The parent class is called a ***base class***.
- The derived class inherits all the capabilities of the ***base class*** but can add features of its own. Note that the base class is unchanged by this process.
- Any class you created can be a base class and any derive class can become a base class to its derived children classes.

- Inheritance is a big payoff since it permits code *reusability*. Once the base class is written and debugged, it needs not to be touched again, but can be adapted to work in different situations. Reusing existing code saves time and money and increases program reliability.

- The diagram below illustrates the concept of inheritance. A base class contains several features such as features A & B. By feature we mean data, property & methods. All derived child classes will inherit Features A & B and can add their own additional features, thus making them more powerful.



- For example supposed we create an Employee Class, which contain standard employee features such as name, id, address, benefits etc.  We can then derive classes for each of the different category of employees in the company, such as managers, scientist, laborers etc.
- The UML illustration below demonstrates this concept:



## 8.1.2 Implementing Basic Inheritance

### Creating the Base Class

- Any class we create can be a base class.
- Note that I will use as a convention of using the prefix **m_** for all private variables of the base class to differentiate them from the variables of the derived class. I will use the prefix **m** for all private variables of the derived class.
- The Syntax and example of requirement in the Derived or SubClass to inherit from a Base class is as follows:

```
'Class Header
Public Class SubClassName

Inherits BaseClassName

    Data Definitions

    Properties Definitions

    Methods

End Class
```

**Example:**

- **Creating a Classes:**

- Example a) - Creating a Derived Class Video from a Base Class Product:

```
Public Class Video
        Inherits Products

'Properties,
'Methods
' Event-Procedures


End Class
```

- Example b) - Creating an Employees class from a Person Class:

```
Public Class Employee
        Inherits Person
'Properties,
'Methods
' Event-Procedures


End Class
```

- Lets look at the following clsPerson class example (Note the UML diagram):

**Example 1 (Base Class):**

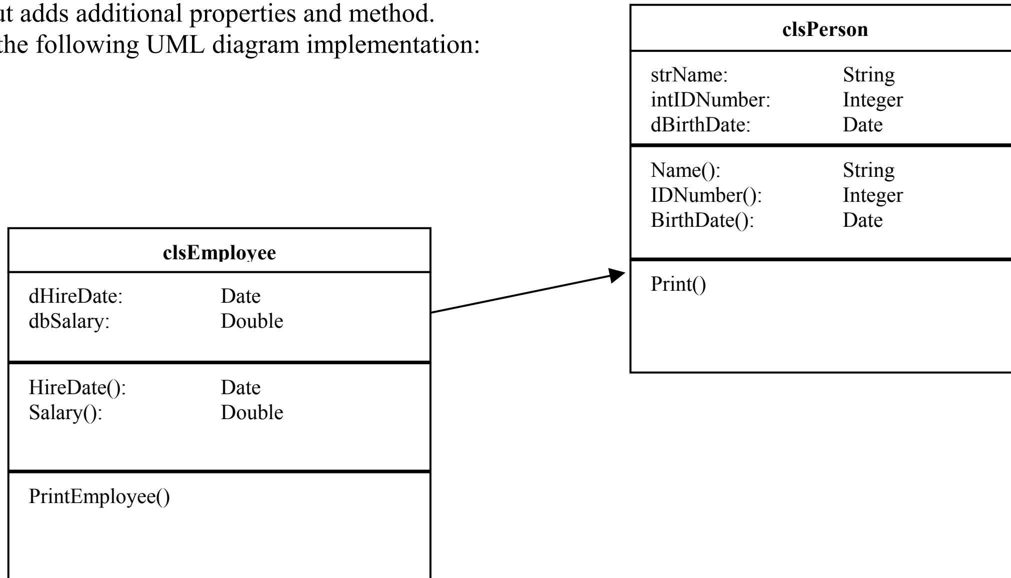
- Declaring the base class:

```
Public Class clsPerson
    Private m_strName As String
    Private m_intIDNumber As Integer
    Private m_dBirthDate As Date


        'Property Declarations
        Public Property Name () As String
            Get
                Return m_strName
            End Get

            Set ( ByVal Value As String )
                m_strName = Value
            End Set
        End Property

        Public Property IDNumber () As Integer
            Get
                Return m_intIDNumber
            End Get

            Set ( ByVal Value As Integer )
                m_intIDNumber = Value
            End Set
        End Property


        Public Property BirthDate () As Date
            Get
                Return m_dBirthDate
            End Get

            Set ( ByVal Value As Date )
                m_dBirthDate = Value
            End Set
        End Property

        'Person Method Declarations
        Public Sub Print ()
          MessageBox.Show("Printing Person Data " & _
                  m_strName & ", " & m_intIDNumber & ", " &  m_dBirthDate)

        End Sub


End Class
```

| **clsPerson** | |
|---|---|
| strName: | String |
| intIDNumber: | Integer |
| dBirthDate: | Date |
| Name(): | String |
| IDNumber(): | Integer |
| BirthDate(): | Date |
| Print() | |

## Creating the Subclass (Derived Class)

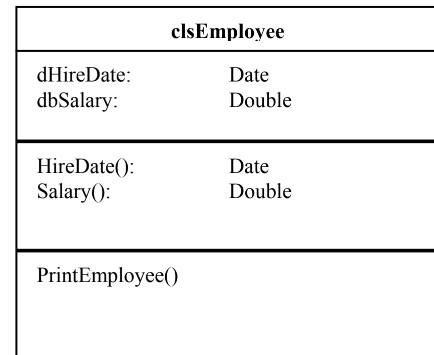
- Using the ***Inherit*** keyword in a class declaration, we can derive other classes from the clsPerson class.
- For example supposed we wished to create an Employee class ***clsEmployee*** as a subclass to ***clsPerson***, which inherits the feature from *clsPerson*, but adds additional properties and method.
- Suppose we want the following UML diagram implementation:




**Example 1 (SubClass):**

- Declaring the SubClas:

```
Public Class clsEmployee
    'Keyword used to implement Inheritance:
    Inherits clsPerson

    '*********************************************************************
    'Class Data or Variable declarations
    Private mdHireDate As String
    Private mdbSalary As Double

    '*********************************************************************
    'Property Procedures
    Public Property HireDate() As String
        Get
            Return mdHireDate
        End Get
        Set(ByVal Value As String)
            mdHireDate = Value
        End Set
    End Property

    Public Property Salary() As Integer
        Get
            Return mdbSalary
        End Get
        Set(ByVal Value As Integer)
            mdbSalary = Value
        End Set
    End Property

    '*********************************************************************
    'Employee Class Method
    Public Sub PrintEmployee()
        MessageBox.Show("Printing Employee Data " _
        & mdHireDate & ", " & mdbSalary)

    End Sub
```



**Using the Base Class & SubClass**

- Now that our subclass is derived from the base class, we can use the properties of the subclass.
- Due to inheritance, objects of the subclass will inherit the functionality of the base class
- For example, the subclass ***clsEmployee*** does not implement the properties *Name*, *BirthDate* and *IDNumber*, but objects of this class will show that *Name*, *BirthDate* and *IDNumber* are property members but they are really not, they are implemented by ***clsPerson*** the base class.
- Note that the private variables m_int*Name*, m_*BirthDate* and m_*IDNumber* will not be accessible by the child class, since they are private. The child or subclass only has access to public members and inherits them directly
- Let's look at a main test program. We will create an object of the base class as well as the subclass in order to demonstrate inheritance.
- ***Main()*** test program:

**Example 1 (Main Program):**

- Driver Program for testing inheritance:

```
'Declare & Create Public Person & Employee Objects
Public objPerson As clsPerson = New clsPerson()
Public objEmployee As clsEmployee = New clsEmployee()


Public Sub Main()

    'Populating Person Object with Data
    With objPerson
        .Name = "Joe Smith"
        .IDNumber = 111
        .BirthDate = #1/2/1965#
    End With

    'Call Person Object Only Method
    objPerson.Print()



    'Populating Employee Object with Data
    With objEmployee
        .Name = "Mary Johnson"
        .IDNumber = 444
        .BirthDate = #4/12/1970#
        .HireDate = #3/9/2004#
        .Salary = 30000
    End With

    'Call Employee Object two available Methods
    objEmployee.Print()
    objEmployee.PrintEmployee()

End Sub
```

**Explanation of Test program:**

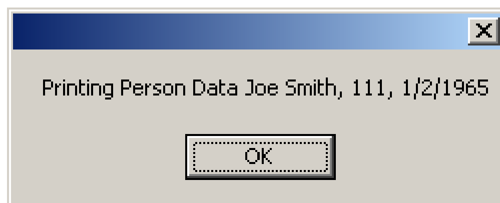
- When we execute the program, the following occurs:

1. **We expose the only two property of the Base Class *objPerson* object, and populate them with data and we call it's *Print()* method:**

```
'Populating Person Object with Data
With objPerson
    .Name = "Joe Smith"
    .IDNumber = 111
    .BirthDate = #1/2/1965#
End With

'Call Person Object Only Method
objPerson.Print()
```

**Results and Explanation:**

- Note that the person object has no access to its derived child's data; it only sees its three properties *Name*, *BirthDate* and *IDNumber*.
- Calling the *objPerson.Print()* method will result printing only the Person object's information as expected:




- There is nothing new here in what we have done so far.

2. **We now populate the SubClass object *objEmployee* and notice that it has a total of 5 properties and two methods. We populate each of the properties and call each of the methods: *Print()* and *PrintEmployee()*:**

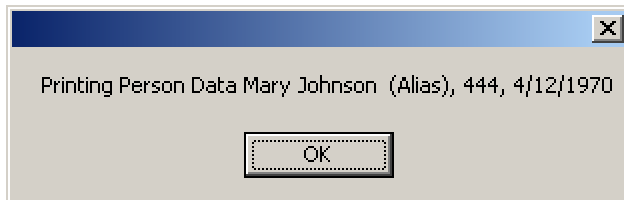
```
'Populating Employee Object with Data
With objEmployee
    .Name = "Mary Johnson"
    .IDNumber = 111
    .BirthDate = #4/12/1970#
    .HireDate = #3/9/2004#
    .Salary = 30000
End With

'Call Employee Object two available Methods
objEmployee.Print()
objEmployee.PrintEmployee()
```

**Results and Explanation:**

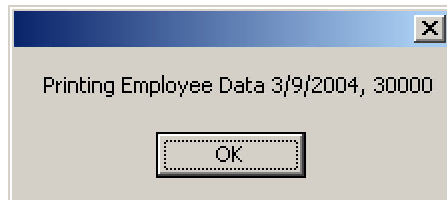
- The derive child objEmployee sees five properties, three properties from the Base Class Person object: *Name*, *BirthDate* and *IDNumber*, and two properties which the clsEmployee class added: *HireDate* & *Salary*.
- We can clearly see that objects of the Employee class inherited the properties *Name*, *BirthDate* and *IDNumber* from Person, and added two of its own *HireDate* & *Salary*.

- We also notice that the derived child, has two methods, one which it inherited from the parent Base class Person and one it added itself.

- Calling the *objEmployee.Print()* method will result printing only the Person object's information as expected:




- Calling the *objEmployee.PrintEmployee()* method will result printing only the Employee object's information as expected:




## Summary:

- We clearly showed that we can inherit all the features of the Base Class and add features of our own in the subclasses.
- We took advantage of the interface and behavior of the Person class and extended it via an Employee class to represent an employee.
- By using an existing Person class we saved development time when creating an Employee class. Another example of ***reusability***!

**Alternate Implementation of the Subclass PrintEmployee Method**

- As we have seen, using the ***Inherit*** keyword will allow us access to the Public Properties & Methods of the base class *clsPerson*.
- If this is the case, there is nothing stopping us from calling the Person Class *Print()* method from within the Employee's Class *PrintEmployee()* method.
- This makes more sense, when we call *Employee.PrintEmployee* in one shot we print both the Base Class *Print()* and the Derived Class *PrintEmployee()*.
- Not only does this makes more sense, but it also represents a real world entity (Employee) since we only make calls to the Employee object
- Let's look at our new implementation of the SubClass ***clsEmployee***.
- Note that we assume the Base Class ***clsPerson***, is the same as the previous example:

**Example 2 (SubClass):**

- Declaring the SubClass:

```
Public Class clsEmployee
    Inherits clsPerson
    '*********************************************************************
    'Class Data or Variable declarations
    Private mdHireDate As String
    Private mdbSalary As Double

    '*********************************************************************
    'Property Procedures
    Public Property HireDate() As String
        Get
            Return mdHireDate
        End Get
        Set(ByVal Value As String)
            mdHireDate = Value
        End Set
    End Property

    Public Property Salary() As Integer
        Get
            Return mdbSalary
        End Get
        Set(ByVal Value As Integer)
            mdbSalary = Value
        End Set
    End Property

    '*********************************************************************
    'Regular Class Methods
    Public Sub PrintEmployee()
        'Call Inherited Print Method to display Base Class values
        Print()

        'Now display Derived Class values
        MessageBox.Show("Printing Employee Data " _
        & mdHireDate & ", " & mdbSalary)

    End Sub
```

| clsEmployee | |
|---|---|
| dHireDate: | Date |
| dbSalary: | Double |
| HireDate(): | Date |
| Salary(): | Double |
| PrintEmployee() | |

## Using the SubClass in a Main Program

- As in the previous Example 1, due to inheritance, objects of the subclass will inherit the functionality of the base class
- This main program is identical to Example 1, with the exception that for the Employee Object created, we only need to call it's PrintEmployee() method, which in turns automatically calls the Base Class Print().
- ***Main()*** test program:

**Example 2 (Main Program):**

- Driver Program for testing inheritance:

```
Module modMainModule

    'Declare & Create Public Person & Employee Objects
    Public objPerson As clsPerson = New clsPerson()
    Public objEmployee As clsEmployee = New clsEmployee()


    Public Sub Main()

        'Populating Person Object with Data
        With objPerson
            .Name = "Joe Smith"
            .IDNumber = 111
            .BirthDate = #1/2/1965#
        End With

        'Call Person Object Only Method
        objPerson.Print()



        'Populating Employee Object with Data
        With objEmployee
            .Name = "Mary Johnson"
            .IDNumber = 111
            .BirthDate = #4/12/1970#
            .HireDate = #3/9/2004#
            .Salary = 30000
        End With

        'Call Employee Object Method
        objEmployee.PrintEmployee()

    End Sub


End Module
```

**Explanation of Example 2 program:**

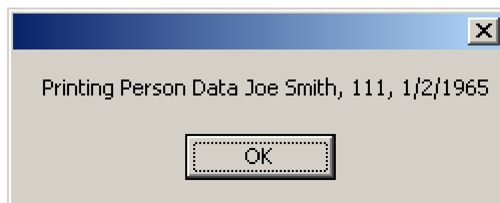
- When we execute the program, the following occurs:

1. **We Populate the only two property of the Base Class *objPerson* object, and populate them with data and we call it's *Print()* method:**

```
'Populating Person Object with Data
With objPerson
    .Name = "Joe Smith"
    .IDNumber = 111
    .BirthDate = #1/2/1965#
End With

'Call Person Object Only Method
objPerson.Print()
```

**Results and Explanation:**

- Note that the person object has no access to its derived child's data; it only sees its three properties *Name*, *BirthDate* and *IDNumber*.
- Calling the *objPerson.Print()* method will result printing only the Person object's information as expected:

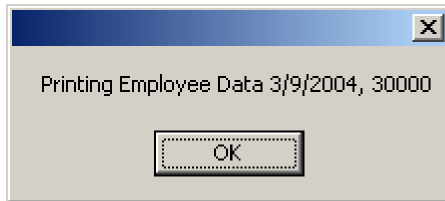



2. **We now populate the SubClass object *objEmployee*. Now we only need to call the *PrintEmployee()* Method:**

```
'Populating Employee Object with Data
With objEmployee
    .Name = "Mary Johnson"
    .IDNumber = 111
    .BirthDate = #4/12/1970#
    .HireDate = #3/9/2004#
    .Salary = 30000
End With

'Call Employee Object Method
objEmployee.PrintEmployee()
```

**Results and Explanation:**

- The derived child object *objEmployee* only needs to call its one *objEmployee.PrintEmployee()* method, which automatically calls its inherit *Print()* method will result printing only the Person object's information first, follow by the derived object's data:




## Summary:

- Since we inherit all the features of the We clearly showed that we have flexibility as to where we call the Base Class public members.
- To better meet the Object-Oriented Programming requirements, it is better to abstract all the Base class implementations from within the Sub Class.

## 8.1.3 Available Access to Base Class Members from SubClasses

### Access Public & Private Members of the Base Class

- In the previous example we saw how the Base Class had NO access to Members of the SubClass.
- More important, we saw how the sub class only had access to Public Members of the Base Class (Public Properties & Methods)
- The rule data encapsulation of Object-Oriented-Programming always hold

***Private data is <u>private</u> and only members of the class have access to it!***

- Therefore derived classes DO NOT have access to their parent's ***Private*** data only to the ***<u>Public</u>*** Interface (*Properties & Methods*)
- So, a derived class cannot directly access the private data, but it does automatically inherit the public interface, thus it seems that the child class contains these parent public interfaces as its own.

### The "Protected" Access Keyword

- In inheritance there is another level of security in the Base Class offered for *SubClasses*. This level is known as Protected Data, using the keyword "**<u>Protected</u>**".
- The Protected keyword means that derived classes are the only ones that can access *protected* members of the base class
- To any other class a variable declared with the keyword Protected is Private. The rule is:

***No other classes other than a derived class have access to a Protected Member!***

- You use *protected* members when you want to give derived classes direct access to these members.
- The moral is that if you are writing a class that you suspect might be used, at any point in the future, as a base class for other classes, then any data or functions that the derived classes might need to access should be made **protected**. This ensures that the class is **"inheritance ready"**
- There is a disadvantage to making members protected. Protected members are less secure because anyone can derive a class and access the protected members. Therefore precautions should be taken
- We will show examples of the Protected Keyword in later sections

### Summary

- The table below is a summary of the basic access specification for classes in general:

| ACCESS SPECIFIER | ACCESSIBLE FROM ITSELF | ACCESSIBLE FROM DERIVED CLASS | ACCESSIBLE FROM OBJECTS OUTSIDE CLASS |
|---|---|---|---|
| **Public** | Yes | Yes | Yes |
| **Protected** | Yes | Yes | No |
| **Private** | Yes | No | No |

# 8.2 Inheritance Concepts

## 8.2.1 Inheritance Features

- In this section we will cover some of the features available via inheritance.
- Inheritance is a powerful tool of VB.NET and contains much functionality. I will only cover the following:

  - Overloading Methods & Properties
  - Overriding Methods & Properties
  - MyBase Keyword
  - MyClass Keyword
  - Level of Inheritance
  - Constructors
  - Protected Scope
  - Abstract Base Class

## 8.2.2 Method Overloading in Inheritance

### General Method Overloading Review

- In past lectures we covered the topic of Method Overloading.
- Method Overloading gave us the ability to implement methods with the same name, as long as their Method Signature is different.
- The Method Signature refers to the number of parameters and return type of a method.
- As long as the numbers of arguments are different, we can create methods having the same name.
- Let's look at an old example of the various valid declarations of the method ***CalculateTotal()***:

  - Using Method Overloading we can declare the following Methods inside a class named Invoice:

```
'No parameters version
Public Sub CalculateTotal ()
        'Code in what ever you desire here!
        decTotal = decSubTotal + (decSubTotal * decTax)
End Sub

'Two parameters version with data type: dec & dec
Public Sub CalculateTotal (ByRef decTotal As Decimal, ByVal decTax As Deciaml)
        decTotal = decSubTotal + (decSubTotal * decTax)
End Sub

'One parameters version with data type: dec
Public Sub CalculateTotal ( decTotal As Decimal)
        decTotal = decSubTotal  * decTax
End Sub

'One parameters version with data type: int
Public Sub CalculateTotal (ByVal intTotal As Integer)
        intTotal = intValue
End Sub

'Two parameters version with data type: int & int
Public Sub CalculateTotal (intTotal As Integer, charName As String)
        intTotal = intValue
        charNamel = charValue
End Sub
```

    - Note that not one of these methods are identical..**ONLY THE NAME!!!!!!**

  - From these declarations, we can make the following calls:

```
objInvoice.CalculateTotal(decTotalCharges)    'Will call the one-paremeter dec version

objInvoice.CalculateTotal(decTotalCharges, decSalesTax)  '2-parameter, 2-dec data types

objInvoice.CalculateTotal()   'No argument version

objInvoice.CalculateTotal(intTotalValue, charClientName)   '2-par, int & char data types

objInvoice.CalculateTotal(intValue)   '1-par, int data type version
```

  - Each of these calls will call the corresponding method that matches its number of parameter & data type

## Implementing Method Overloading in Inheritance

- Method overloading can be applied to our Derived or SubClasses.
- In other words, we can overload a Base Class Method thus extending and providing another implementation of the inherited method.
- As long as the names are the same but the number of parameters are different, we can overload a base class method
- Note that the original Base class method is still available, but in the child class we extended it by adding another one the performs some other implementation of the base class method.
- This is the beauty of inheritance, not only can we inherit, but we can extend the features currently available by the Base Class
- Lets look at another version of the previous example where we will overload the *Print()* method of the Base Class by adding a Print(int X) method in the derived class that will Print the Base Class data X times, and overload the Name Property to add a comment to the Name string.

## Example 3 – Overloading Methods

### Creating the Base Class

- Re-using the clsPerson class from the previous example:

**Example 3 (Base-Class):**

- Declaring the base class:

```
Public Class clsPerson
    '*********************************************************************
    'Class Data or Variable declarations
    Private m_strName As String
    Private m_intIDNumber As Integer
    Private m_dBirthDate As Date

    '*********************************************************************
    'Property Procedures
    Public Property Name() As String
        Get
            Return m_strName
        End Get
        Set(ByVal Value As String)
            m_strName = Value
        End Set
    End Property

    Public Property IDNumber() As Integer
        Get
            Return m_intIDNumber
        End Get
        Set(ByVal Value As Integer)
            m_intIDNumber = Value
        End Set
    End Property

    Public Property BirthDate() As Date
        Get
            Return m_dBirthDate
        End Get
        Set(ByVal Value As Date)
            m_dBirthDate = Value
        End Set
    End Property

    '*********************************************************************
    'Regular Class Methods
    Public Sub Print()
        MessageBox.Show("Printing Person Data " _
        & m_strName & ", " & m_intIDNumber & ", " & _
        m_dBirthDate)

    End Sub
```

| clsPerson | |
|---|---|
| strName: | String |
| intIDNumber: | Integer |
| dBirthDate: | Date |
| Name(): | String |
| IDNumber(): | Integer |
| BirthDate(): | Date |
| Print() | |

### Overloading the Print Method using the OverLoads Keyword

- We create the clsEmployees class and as usual we use the ***Inherit*** keyword in a class declaration to inherit from the *clsPerson* Class.
- In order to implement method overloading we need to use the keyword Overload in the declaration of the method or property.
- Using the keyword **Overload**, we add another Name Property which takes as argument a string representing a comment that will be added to the Name string.
- Using the keyword Overload, we overload the Base Class *Print()* method by adding another Method named *Print(X)* which takes one argument.
- Lets look at the derived class *clsEmployee*:

**Example 3 (SubClass):**

- Declaring the SubClas:

```
Public Class clsEmployee
    Inherits clsPerson
    '*********************************************************************
    'Class Data or Variable declarations
    Private mdHireDate As String
    Private mdbSalary As Double

    '*********************************************************************
    'Property Procedures
    Public Property HireDate() As String
        Get
            Return mdHireDate
        End Get
        Set(ByVal Value As String)
            mdHireDate = Value
        End Set
    End Property

    Public Property Salary() As Integer
        Get
            Return mdbSalary
        End Get
        Set(ByVal Value As Integer)
            mdbSalary = Value
        End Set
    End Property

'Overloading the Base Class Name Property
    Public Overloads Property Name(ByVal strComment As String) As String
        Get
            Return Name
        End Get
        Set(ByVal Value As String)
            'Add the Comment to the end of the name
            Name = Value & "  (" & strComment & ")"

        End Set
    End Property
```

| clsEmployee | |
|---|---|
| dHireDate: | Date |
| dbSalary: | Double |
| HireDate(): | Date |
| Salary(): | Double |
| Name(String): | String |
| Print(X) | |
| PrintEmployee() | |

**Example 3 (SubClass-(Cont)):**

- Declaring the SubClass Methods:

```
    '**********************************************************************
    'Regular Class Methods

    'Overloaded Base Class Method
    Public Overloads Sub Print(ByVal intNumberOfPrints As Integer)
        Dim i As Integer

        For i = 1 To intNumberOfPrints
            MessageBox.Show("Multiple Print Jobs for: " _
            & Name & ", " & IDNumber & ", " & _
            BirthDate)
        Next

    End Sub

    Public Sub PrintEmployee()
        'Call Print() Method to display Base Class Data
        Print()

        'Display Derived Class Data
        MessageBox.Show("Printing Employee Data " _
        & mdHireDate & ", " & mdbSalary)

    End Sub

End Class
```

**Using the SubClass and Calling the Overloaded Property & Method**

- Now let's look at the driver program.
- In this example we create two objects of the ***clsEmployee*** class. We will no longer need to create objects of the Base Class, unless necessary, since the derived class objects contain everything from the base and more.
- We assign values to the first Employee Object using the standard Properties inherited by the Base Class: *Name*, *BirthDate* and *IDNumber*, those provided by the derived class: *HireDate* & *Salary*.
- We call the first Employee Object PrintEmployee Method to print both the Base Class data and Derived Class data.
- In the second Employee Object, we assign values to only two of the properties inherited by the Base Class: *BirthDate* and *IDNumber*, we chose NOT to use the inherited property *Name*, but decided to use the properties provided by the derived class: Overloaded Property *Name(X)*, and the regular *HireDate* & *Salary*
- In the second Employee object we call the PrintEmployee() method to print both Base & Derived Class data and in addition we call the overloaded method Print(X) to print only the Base Class data X times.
- ***Main()*** test program:

**Example 3 (Main Program):**

- Driver Program for testing inheritance:

```
Module modMainModule

    'Declare & Create Public Person & Employee Objects
    Public objEmployee1 As clsEmployee = New clsEmployee()
    Public objEmployee2 As clsEmployee = New clsEmployee()


    Public Sub Main()

        'Populating Person Object with Data
        With objEmployee1
            .Name = "Joe Smith"
            .IDNumber = 111
            .BirthDate = #1/2/1965#
            .HireDate = #5/23/2004#
            .Salary = 50000
        End With

        'Call Employee Object Method
        objEmployee1.PrintEmployee()



        'Populating Employee2 Object with Data
        With objEmployee2
            'Assign Overloaded Property
            .Name("Alias") = "Mary Johnson"
            .IDNumber = 444
            .BirthDate = #4/12/1970#
            .HireDate = #3/9/2004#
            .Salary = 30000
        End With

        'Call Employee Class PrintEmployee method
        objEmployee2.PrintEmployee()

        'Call Overloaded PrintPerson method
        objEmployee2.Print(3)

    End Sub

End Module
```

**Explanation of Test program:**

- When we execute the program, the following occurs:

1. **We create two Employee Objects:**

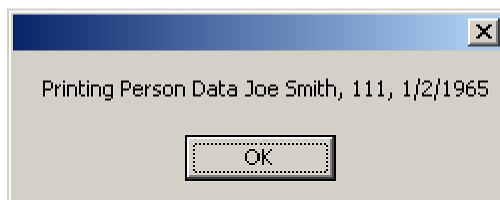
```
'Declare & Create Public Person & Employee Objects
Public objEmployee1 As clsEmployee = New clsEmployee()
Public objEmployee2 As clsEmployee = New clsEmployee()
```

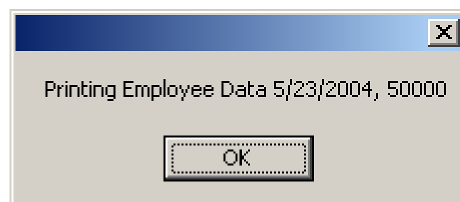
2. **We populate the First Object using the Inherited properties from the Base Class and added properties of the Employee Class and we call it's PrintEmployee() Method to print Base & Derived Class data:**

```
'Populating Person Object with Data
With objEmployee1
    .Name = "Joe Smith"
    .IDNumber = 111
    .BirthDate = #1/2/1965#
    .HireDate = #5/23/2004#
    .Salary = 50000
End With

'Call Employee Object Method
objEmployee1.PrintEmployee()
```

**Results and Explanation:**

- When populating *Name*, *BirthDate* and *IDNumber* we are using the Base Class Properties, and when populating *HireDate* & *Salary* we are using the Derived Class properties.
- Calling the *objEmployee1.PrintEmployee()* method will result printing both the Base Class data and Derived Class data. This is how it was programmed:



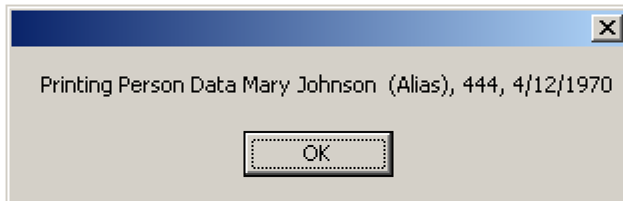



3. **We now populate the Second Object using two of the Inherited Properties from the Base Class and added properties of the Employee Class including the Overloaded Name property. In addition we call it's PrintEmployee() Method to print Base & Derived Class data:**

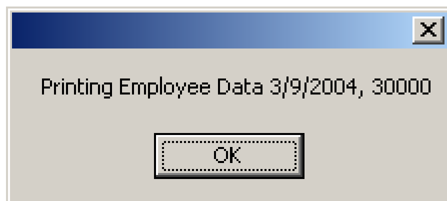
```
'Populating Employee2 Object with Data
With objEmployee2
    'Assign Overloaded Property
    .Name("Alias") = "Mary Johnson"
    .IDNumber = 444
    .BirthDate = #4/12/1970#
    .HireDate = #3/9/2004#
    .Salary = 30000
End With

'Call Employee Class PrintEmployee method
objEmployee2.PrintEmployee()
```

**Results and Explanation:**

- In this object we chose to only populate the populate from the Base Class the *BirthDate* and *IDNumber*. For the derived class we populate *HireDate* & *Salary* but in addition populate the **Overloaded** Name(X) Property and send a text string as an argument.
- We then call the *objEmployee2.PrintEmployee()* method will result printing both the Base Class data and Derived Class data:




- Calling the *objEmployee.PrintEmployee()* method will result printing only the Employee object's information as expected:



**4. We now call the Overloaded Print(X) method to print the Base class data X times:**

```
'Call Overloaded PrintPerson method
objEmployee2.Print(3)
```

**Results and Explanation:**

- We then call the *objEmployee2.Print(3)* method will are actually calling the Employee Class Overloaded representation of the Base Class. The Base Class data will print 3 times:

**Summary:**

- We clearly showed that we can not only inherit all the features of the Base Class and add features of our own in the subclasses but also extend the Base Class features by **Overloading** them and extending them to perform more functionalities.

## 8.2.3 Method Overriding

### Introduction

- In the previous section we learned *Method Overloading*. Overloading allowed us to extend the functionality of a Base class Method or Property by adding a new version in the Derived Class with the same name, but as long as the parameter list is different.
- The key point to *Overloading* is that we kept the original functionality of the base and just added a new or additional functionality in the child or *SubClass*.
- Now let's supposed we want NOT just extend an implementation of the base class, but change or completely replace a functionality of a method or property.
- This is where ***Method Overriding*** comes in to play.
- Method Overriding gives us the ability to completely replace the implementation of a base class method or property with a NEW or overridden method in the SubClass with the Same Name and signature.
- The key point here is that we are replacing! The new method has the same signature (Name, # of parameters, return type etc).

### Implementing Method Overriding

- To implement Method Overriding we need to use two keyword: ***Overridable*** & ***Overrides***
- To implement we first need to realize that we just can't simply override a Base Class. The base class needs to give us permission to do so, in other words the Method or Property in the Base Class must grant this feature. This is where the keyword ***Overridable*** is used.

#### Overridable keyword

- The ***Overridable*** must be stated in the Base Class on every Method or Property in which the Base Class allows the Derived Classes to override.
- The idea here is that the Base Class is in control of which Methods and Property a Derived class can override.

#### Overrides keyword

- Once a Property or Method has the *Overridable* keyword, the derived class can override the Method/Property using the keyword ***Overrides***. This keyword tells the SubClass that this Method/Property is to override the one in the Parent or Base Class.
- The overridden method in the Base class will not execute at all via the Sub Class. Only the new version will execute.
- Now don't get confuse by this statement. Note that we are saying that the overridden method in the derive class will run and not the one in the base class. But this is only when we are trying to call the method from and object of the child or derived class that the new one executes. You can still run the original but only if you create an object of the Base Class as expected.

## Example 4 – Overriding Property & Methods

- Lets look at another version of the previous example where this time we will override the *BirthDate* Property and the *Print()* method of the Base Class by replacing it with a NEW version of *BirthDate* and *Print()* method in the derived class.
- In this example we will prove the following:
  - Method Overridden executes and not Base Class Method
  - Original Method in Base class can be accessed but only by Base Class Objects
  - Throwing an Exception

### Creating the Base Class

- Using the keyword ***Overridable*** we allow the *Birthdate* & *Print()* method to be overridden:

**Example 4 (Base-Class):**

- Declaring the base class:

```
Option Explicit On
Public Class clsPerson
    '*********************************************************************
    'Class Data or Variable declarations
    Private m_strName As String
    Private m_intIDNumber As Integer
    Private m_dBirthDate As Date

    '*********************************************************************
    'Property Procedures
    Public Property Name() As String
        Get
            Return m_strName
        End Get
        Set(ByVal Value As String)
            m_strName = Value
        End Set
    End Property

    Public Property IDNumber() As Integer
        Get
            Return m_intIDNumber
        End Get
        Set(ByVal Value As Integer)
            m_intIDNumber = Value
        End Set
    End Property
    'We allow Property to be overridden
    Public Overridable Property BirthDate() As Date
        Get
            Return m_dBirthDate
        End Get
        Set(ByVal Value As Date)
            m_dBirthDate = Value
        End Set
    End Property


    '*********************************************************************
    'Regular Class Methods

    'We allow Method to be overridden
    Public Overridable Sub Print()
        MessageBox.Show("Printing BASE CLASS Person Data " _
        & m_strName & ", " & m_intIDNumber & ", " & _
        m_dBirthDate)
    End Sub
```

| clsPerson | |
|---|---|
| strName: | String |
| intIDNumber: | Integer |
| dBirthDate: | Date |
| Name(): | String |
| IDNumber(): | Integer |
| BirthDate(): | Date |
| Print() | |

### Creating Derived Class & Overriding the BirthDate Property

- We create the clsEmployees class and as usual we use the ***Inherit*** keyword in a class declaration to inherit from the *clsPerson* Class.
- We create a New *BirthDate* Property inside the clsEmployee Class and we use the keyword ***Overrides*** in the declaration of the property to always use this *BirthDate* Property instead of the Base *BirthDate* version.
- This new implementation of *BirthDate*, implements a new policy within the company that every employee must be at least 16 years old.   If an employee is under 16, we need to raise a flag or a warning.
- I implemented this warning by Throwing an Exception.  This will help us review Throwing Exceptions
- You will also notice that we are FORCED to create a new *Private* Variable ***mdBirthDate*** in order to store the New BirthDate Data
- This is very important and difficult to understand.
- Why are we force?  Because the Base Class ***m_dBirthDate*** is private an inaccessible.   More important we cannot call the Base Class Public BirthDate Property to access ***m_dBirthDate*** from within our NEW version of BirthDate Property since the compiler will get confused with which Birthdate are you referring to, the new one or old one, it cannot tell, once overridden the one in the Base Class is not recognized from within the child.  This is important…more on this later….
- Lets look at the derived class *clsEmployee*:

**Example 4 (SubClass):**

- Declaring the SubClas:

```
Public Class clsEmployee
    Inherits clsPerson
    '*********************************************************************
    'Class Data or Variable declarations
    Private mdHireDate As String
    Private mdbSalary As Double
    'Create a new private variable to store new birthdate information
    Private mdBirthDate As Date

    '*********************************************************************
    'Property Procedures
    Public Property HireDate() As String
        Get
            Return mdHireDate
        End Get
        Set(ByVal Value As String)
            mdHireDate = Value
        End Set
    End Property

    Public Property Salary() As Integer
        Get
            Return mdbSalary
        End Get
        Set(ByVal Value As Integer)
            mdbSalary = Value
        End Set
    End Property

    'We Override the Birthdate Property
    Public Overrides Property BirthDate() As Date
        Get
            Return mdBirthDate
        End Get
        Set(ByVal Value As Date)
            'Test to verify that Employee meets age requirement
            If DateDiff(DateInterval.Year, Value, Now()) >= 16 Then
                mdBirthDate = Value
            Else
Throw New System.Exception("Under Age Employee, an Employee must be 16 Years old")
            End If

        End Set
    End Property
```

| clsEmployee | |
|---|---|
| dHireDate: | Date |
| dbSalary: | Double |
| HireDate(): | Date |
| Salary(): | Double |
| Name(String): | String |
| Print(X) | |
| PrintEmployee() | |

### Overriding the Print() Method

- Now we override the Print() Method using the keyword ***Overrides***.
- This is the new version that will execute instead of the one written in the Base Class.
- Note that the code in this new method is simply printing the Base Class data as the Base Class Counterpart. I am doing this for teaching purpose only, the point you need to keep in mind is that it is this NEW Print() that will execute Not the Base Class Print().
- Lets continue our implementation of the class *clsEmployee*:

**Example 4 (SubClass-(Cont)):**

- Declaring the SubClass Methods:

```
    '*********************************************************************
    'Regular Class Methods

   'NEW Overriden Method
    Public Overrides Sub Print()

        'Display Inherited Base Class Properties, NEW Overriden BirthDate Property
        MessageBox.Show("Printing NEW IMPROVED Employee Data " _
        & Name & ", " & IDNumber & ", " & BirthDate)

    End Sub

    Public Sub PrintEmployee()
        'Call Overriden Print() Method to display Base Class Data
        Print()

        'Display Derived Class Data
        MessageBox.Show("Printing Employee Data " _
        & mdHireDate & ", " & mdbSalary)

    End Sub
End Class
```

### Proving our theory by Calling the Overridden Property & Method

- Now let's look at the driver program.
- In this example we create three objects, one of the Base Class ***clsPerson*** and two of the ***clsEmployee*** class.
- The will use the Base Class Object simply to prove that the Print() Method of this object is still valid for Person Objects, but NOT for the Derived Classes. We will do this by assigning values to this object and calling the Print() method.
- In the first Employee object we will assign values using the standard Properties inherited by the Base Class: *Name* and *IDNumber*, (Note that *Birthdate* is overridden and no longer inherited) those provided by the derived class: *BirthDate (Overridden)*, *HireDate* & *Salary*.
- We call the first Employee Object *PrintEmployee()* Method to print both the Base Class data and Derived Class data.
- In the second Employee Object perform the same operations.
- ***Main()*** test program:

**Example 4 (Main Program):**

- Driver Program for testing inheritance:

```
Option Explicit On
Module modMainModule
    'Declare & Create Public Person & Employee Objects
    Public objEmployee1 As clsEmployee = New clsEmployee()
    Public objEmployee2 As clsEmployee = New clsEmployee()
    Public objPerson As clsPerson = New clsPerson()

    Public Sub Main()

        'Populating Person Object with Data
        With objPerson
            .Name = "Frank Lee"
            .IDNumber = 123
            .BirthDate = #4/23/1968#
        End With

        'Call Person Print Method to Execute Base Class Print()
        objPerson.Print()

        'Populating Employee Object with Data
        '(Note that BirthDate Property used is actually the overridden Version)
        With objEmployee1
            .Name = "Joe Smith"
            .IDNumber = 111
            .BirthDate = #1/2/1965#
            .HireDate = #5/23/2004#
            .Salary = 50000
        End With

        'Call Employee Print Method which Executes embedded Overridden Print()
        objEmployee1.PrintEmployee()

        'Populating Employee Object with Data
        '(Note that BirthDate Property used is actually the overridden Version)
        '(Also note that BirthDate = Date < 16, thus Error will be raised)
        With objEmployee2
            .Name = "Mary Johnson"
            .IDNumber = 444
            .BirthDate = #4/12/1989#
            .HireDate = #5/23/2004#
            .Salary = 30000
        End With

        'Call Employee Print Method which Executes embedded Overridden Print()
        objEmployee2.PrintEmployee()

    End Sub

End Module
```

## Explanation & Results of Main Program:

- When we execute the program, the following occurs:

**1. We create one Person Object and two Employee Objects:**

```
'Declare & Create Public Person & Employee Objects
Public objEmployee1 As clsEmployee = New clsEmployee()
Public objEmployee2 As clsEmployee = New clsEmployee()
Public objPerson As clsPerson = New clsPerson()
```

**2. We populate the Base Class Object data and call it's Print() Method to print Base Class data:**

```
'Populating Person Object with Data
With objPerson
    .Name = "Frank Lee"
    .IDNumber = 123
    .BirthDate = #4/23/1968#
End With

'Call Person Print Method to Execute Base Class Print()
objPerson.Print()
```

**Results and Explanation:**

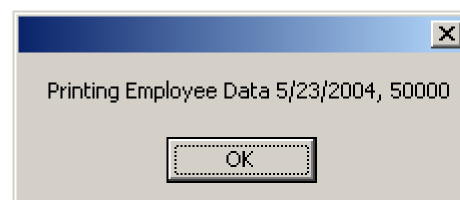
- Note that the Print() method for the Base is still operational as the resultant message box indicates, but only for Base Class Objects:

**3. We populate the first Employee Object using the Inherited properties from the Base Class, the Overridden Birthdate Property of the derived class and the remaining properties added by the Employee Class. In addition and we call it's PrintEmployee() Method to print the Overridden Base Class Print() method & Derived Class data:**

```
'Populating Employee Object with Data
'(Note that BirthDate Property used is actually the overridden Version)
With objEmployee1
    .Name = "Joe Smith"
    .IDNumber = 111
    .BirthDate = #1/2/1965#
    .HireDate = #5/23/2004#
    .Salary = 50000
End With

'Call Employee Print Method which Executes embedded Overridden Print()
objEmployee1.PrintEmployee()
```

**Results and Explanation:**

- Note that the BirthDate Property used here is the Overridden Property not the one from the Base. We will prove this in the following set of code.
- Also note that it is the NEW Overridden Print() method that is executing not the Base Class Print():

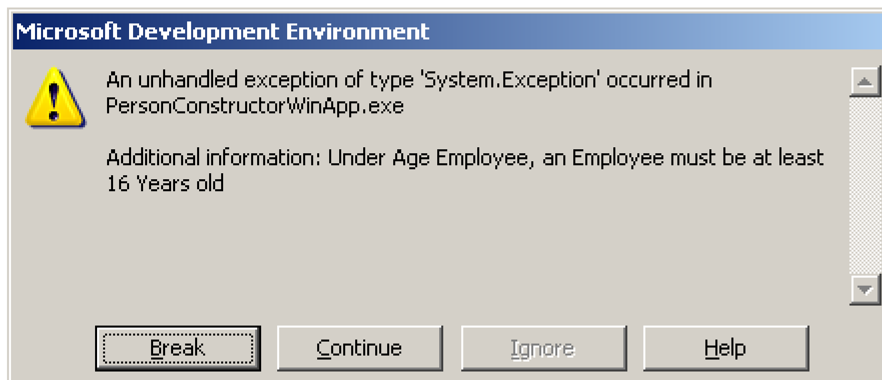


4. **We now populate the Second Object using two of the Inherited Properties from the Base Class, the Overridden BirthDate properties of the Employee Class and the other added Employee Class properties (Salary & HiredDate). In addition we call it's PrintEmployee() Method to print Overridden Base Class Print() method and Derived Class data:**

```
'Populating Employee Object with Data
'(Note that BirthDate Property used is actually the overridden Version)
'(Also note that BirthDate = Date < 16, thus Error will be raised)
With objEmployee2
    .Name = "Mary Johnson"
    .IDNumber = 444
    .BirthDate = #4/12/1989#
    .HireDate = #5/23/2004#
    .Salary = 30000
End With

'Call Employee Print Method which Executes embedded Overridden Print()
objEmployee2.PrintEmployee()
```

**Results and Explanation:**

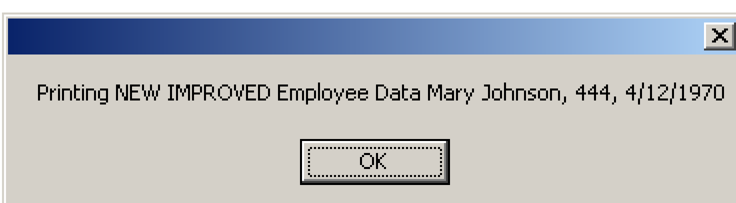
- In this object we populate the  populate from the Base Class the *Name* and *IDNumber*.  For the derived class we populate the Overridden *BirthDate* Property, *HireDate* & *Salary*.
- Remember that the NEW *BirthDate* Property has code that will test to make sure that the employee is over 16 years of age.  Yet the value chosen for the *BirthDate* Property is a year which will indicates that the employee is under 16, therefore an Exception is thrown by our code.
- Since our code contain no Error Handling Code (Try-Catch-Finally Statement) the program will stop execution:

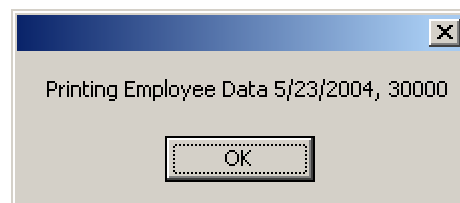



- If the data chosen would have not made the employee under 16, the program would have proceeded to the next code which calls the *objEmployee.PrintEmployee()* method to call the Overriden Print() method and Derived Class data.
- Supposed we would have chosen 1970 as the Birth date year for the Employee Object as follows (NOTE I am changing the date):

```
With objEmployee2
    .Name = "Mary Johnson"
    .IDNumber = 444
    .BirthDate = #4/12/1970#
    .HireDate = #5/23/2004#
    .Salary = 30000
End With

'Call Employee Print Method which Executes embedded Overridden Print()
objEmployee2.PrintEmployee()
```

- Note that it is the NEW Overridden Print() method that is executing not the Base Class Print():



## Example 5 – Example 4 with Error Handling (Overriding Property & Methods Cont)

- In our previous Example 4 we clearly showed how Method Overriding works.
- But our example raised and Exception using the Throw Keyword.  This means that we need to add error handling code using the ***Try-Catch-Finally*** Statement in order to prevent the program from stopping and informing the user during execution.
- In this example we will do the following:
  - Add error handling methods to Example 4 to trap the *BirthDate* Property Generated Exception
  - The Error Handling code will reside in the Main Program.

### Creating the Base Class

- Same as Example 4

### Creating Derived Class, Overriding the BirthDate Property & Print() Method

- Same as Example 4.

### Main Program with Error Handling Code

- Ok the Main program is still the same, but this time we will add a ***Try-Catch-Finally*** statement to trap and handle the error.
- ***Main()*** test program:

**Example 2 (Main Program):**

- Driver Program for testing inheritance:

```
Option Explicit On
Module modMainModule
    'Declare & Create Public Person & Employee Objects
    Public objEmployee1 As clsEmployee = New clsEmployee()
    Public objEmployee2 As clsEmployee = New clsEmployee()
    Public objPerson As clsPerson = New clsPerson()

    Public Sub Main()
        'Begin Error Trapping section
        Try
            'Populating Person Object with Data
            With objPerson
                .Name = "Frank Lee"
                .IDNumber = 123
                .BirthDate = #4/23/1968#

            End With

            'Call Person Print Method to Execute Base Class Print()
            objPerson.Print()


            'Populating Employee Object with Data
            '(Note that BirthDate Property used is actually the overridden Version)
            With objEmployee1
                .Name = "Joe Smith"
                .IDNumber = 111
                .BirthDate = #1/2/1965#
                .HireDate = #5/23/2004#
                .Salary = 50000
            End With

            'Call Employee Print Method which Executes embedded Overridden Print()
            objEmployee1.PrintEmployee()


            'Populating Employee Object with Data
            '(Note that BirthDate Property used is actually the overridden Version)
            '(Also note that BirthDate = Date < 16, thus Error will be raised)
            With objEmployee2
                .Name = "Mary Johnson"
                .IDNumber = 444
                .BirthDate = #4/12/1989#
                .HireDate = #5/23/2004#
                .Salary = 30000
            End With

            'Call Employee Print Method which Executes embedded Overridden Print()
            objEmployee2.PrintEmployee()

            'End Error Trapping section & Begin Error Handling Section
        Catch objException As Exception
            MessageBox.Show(objException.Message)
        End Try

    End Sub
```

## Explanation & Results of Main Program:

- When we execute the program, the following occurs:

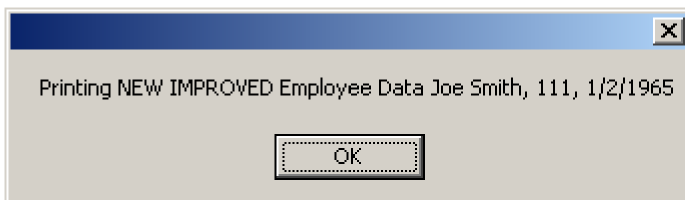
1. **We Create the three Objects As in Example 4**

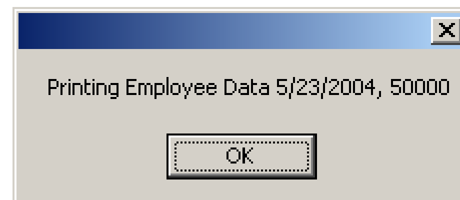
2. **We populate the Base Class Object data and call it's Print() Method to print Base Class data As in Example 4 with the same results:**

   **Results and Explanation:**
   - Note that the Print() method for the Base is still operational as the resultant message box indicates, but only for Base Class Objects:

3. **We populate the first Employee Object using the Inherited properties from the Base Class, the Overridden Birthdate Property of the derived class and the remaining properties added by the Employee Class. In addition and we call it's PrintEmployee() Method to print the Overridden Base Class Print() method & Derived Class data. Same results as Example 4:**

   **Results and Explanation:**
   - Note that the BirthDate Property used here is the Overridden Property not the one from the Base. We will prove this in the following set of code.
   - Also note that it is the NEW Overridden Print() method that is executing not the Base Class Print():



4. **We now populate the Second Object Same as Example 4, but since we have Error handling method we will trap and handle the error appropriately:**

**Results and Explanation:**

- In this case when we populate the NEW *BirthDate* Property with an age is under 16; the Exception thrown is trapped by the ***Try-Catch-Finally*** statement **Catch Block** and handled appropriately by displaying the Exception Object Message Property.

```
'Begin Error Trapping section
Try
    'Populating Person Object with Data
    --------
    'Call Person Print Method to Execute Base Class Print()
    --------


    'Populating Employee Object with Data
    '(Note that BirthDate Property used is actually the overridden Version)
    --------

    'Call Employee Print Method which Executes embedded Overridden Print()
    --------

    'Populating Employee Object with Data
    '(Note that BirthDate Property used is actually the overridden Version)
    '(Also note that BirthDate = Date < 16, thus Error will be raised)
    With objEmployee2
        .Name = "Mary Johnson"
        .IDNumber = 444
        .BirthDate = #4/12/1989#
        .HireDate = #5/23/2004#
        .Salary = 30000
    End With

    'Call Employee Print Method which Executes embedded Overridden Print()
    objEmployee2.PrintEmployee()

    'End Error Trapping section & Begin Error Handling Section
Catch objException As Exception
    MessageBox.Show(objException.Message)
End Try
```

- Also note that the code the follows the error is NOT executed:

```
'Call Employee Print Method which Executes embedded Overridden Print()
objEmployee2.PrintEmployee()
```

- This is because immediately after the Exception is thrown, the program execution JUMPS to the Catch Block to handle the error.
- This is what we want anyway, we don't want to print this employee, she is under aged!!!