Objects and Classes in Visual Basic

Visual Studio 2015

An *object* is a combination of code and data that can be treated as a unit. An object can be a piece of an application, like a control or a form. An entire application can also be an object.

When you create an application in Visual Basic, you constantly work with objects. You can use objects provided by Visual Basic, such as controls, forms, and data access objects. You can also use objects from other applications within your Visual Basic application. You can even create your own objects and define additional properties and methods for them. Objects act like prefabricated building blocks for programs — they let you write a piece of code once and reuse it over and over.

This topic discusses objects in detail.

Objects and Classes

Each object in Visual Basic is defined by a *class*. A class describes the variables, properties, procedures, and events of an object. Objects are instances of classes; you can create as many objects you need once you have defined a class.

To understand the relationship between an object and its class, think of cookie cutters and cookies. The cookie cutter is the class. It defines the characteristics of each cookie, for example size and shape. The class is used to create objects. The objects are the cookies.

You must create an object before you can access its members.

To create an object from a class

1. Determine from which class you want to create an object.

2. Write a Dim Statement (Visual Basic) to create a variable to which you can assign a class instance. The variable should be of the type of the desired class.

```
Dim nextCustomer As customer
```

3. Add the New Operator (Visual Basic) keyword to initialize the variable to a new instance of the class.

```
Dim nextCustomer As New customer
```

4. You can now access the members of the class through the object variable.

```
nextCustomer.accountNumber = lastAccountNumber + 1
```

Note

Whenever possible, you should declare the variable to be of the class type you intend to assign to it. This is called *early binding*. If you do not know the class type at compile time, you can invoke *late binding* by declaring the variable to be of the Object Data Type. However, late binding can make performance slower and limit access to the run-time object's members. For more information, see Object Variable Declaration (Visual Basic).

Multiple Instances

Objects newly created from a class are often identical to each other. Once they exist as individual objects, however, their variables and properties can be changed independently of the other instances. For example, if you add three check boxes to a form, each check box object is an instance of the CheckBox class. The individual CheckBox objects share a common set of characteristics and capabilities (properties, variables, procedures, and events) defined by the class. However, each has its own name, can be separately enabled and disabled, and can be placed in a different location on the form.

Object Members

An object is an element of an application, representing an *instance* of a class. Fields, properties, methods, and events are the building blocks of objects and constitute their *members*.

Member Access

You access a member of an object by specifying, in order, the name of the object variable, a period (.), and the name of the member. The following example sets the Text property of a Label object.

```
warningLabel.Text = "Data not saved"
```

IntelliSense Listing of Members

IntelliSense lists members of a class when you invoke its List Members option, for example when you type a period (.) as a member-access operator. If you type the period following the name of a variable declared as an instance of that class, IntelliSense lists all the instance members and none of the shared members. If you type the period following the class name itself, IntelliSense lists all the shared members and none of the instance members. For more information, see Using IntelliSense.

Fields and Properties

Fields and *properties* represent information stored in an object. You retrieve and set their values with assignment

statements the same way you retrieve and set local variables in a procedure. The following example retrieves the Width property and sets the ForeColor property of a Label object.

```
Dim warningWidth As Integer = warningLabel.Width
warningLabel.ForeColor = System.Drawing.Color.Red
```

Note that a field is also called a *member variable*.

Use property procedures when:

- You need to control when and how a value is set or retrieved.
- The property has a well-defined set of values that need to be validated.
- Setting the value causes some perceptible change in the object's state, such as an `IsVisible` property.
- Setting the property causes changes to other internal variables or to the values of other properties.
- A set of steps must be performed before the property can be set or retrieved.

Use fields when:

- The value is of a self-validating type. For example, an error or automatic data conversion occurs if a value other than **True** or **False** is assigned to a **Boolean** variable.
- Any value in the range supported by the data type is valid. This is true of many properties of type **Single** or **Double**.
- The property is a **String** data type, and there is no constraint on the size or value of the string.
- For more information, see Property Procedures (Visual Basic).

Methods

A *method* is an action that an object can perform. For example, Add is a method of the ComboBox object that adds a new entry to a combo box.

The following example demonstrates the Start method of a Timer object.

```
Dim safetyTimer As New System.Windows.Forms.Timer
safetyTimer.Start()
```

Note that a method is simply a *procedure* that is exposed by an object.

For more information, see Procedures in Visual Basic.

Events

An event is an action recognized by an object, such as clicking the mouse or pressing a key, and for which you can write code to respond. Events can occur as a result of a user action or program code, or they can be caused by the system. Code that signals an event is said to *raise* the event, and code that responds to it is said to *handle* it.

You can also develop your own custom events to be raised by your objects and handled by other objects. For more information, see Events (Visual Basic).

Instance Members and Shared Members

When you create an object from a class, the result is an instance of that class. Members that are not declared with the Shared (Visual Basic) keyword are *instance members*, which belong strictly to that particular instance. An instance member in one instance is independent of the same member in another instance of the same class. An instance member variable, for example, can have different values in different instances.

Members declared with the **Shared** keyword are *shared members*, which belong to the class as a whole and not to any particular instance. A shared member exists only once, no matter how many instances of its class you create, or even if you create no instances. A shared member variable, for example, has only one value, which is available to all code that can access the class.

Accessing Nonshared Members

To access a nonshared member of an object

1. Make sure the object has been created from its class and assigned to an object variable.

```
Dim secondForm As New System.Windows.Forms.Form
```

2. In the statement that accesses the member, follow the object variable name with the *member-access operator* (**.**) and then the member name.

```
secondForm.Show()
```

Accessing Shared Members

To access a shared member of an object

- Follow the class name with the *member-access operator* (**.**) and then the member name. You should always access a **Shared** member of the object directly through the class name.

```
MsgBox("This computer is called " & Environment.MachineName)
```

- If you have already created an object from the class, you can alternatively access a **Shared** member through the object's variable.

Differences Between Classes and Modules

The main difference between classes and modules is that classes can be instantiated as objects while standard modules cannot. Because there is only one copy of a standard module's data, when one part of your program changes a public variable in a standard module, any other part of the program gets the same value if it then reads that variable. In contrast, object data exists separately for each instantiated object. Another difference is that unlike standard modules, classes can implement interfaces.

Note

When the **Shared** modifier is applied to a class member, it is associated with the class itself instead of a particular instance of the class. The member is accessed directly by using the class name, the same way module members are accessed.

Classes and modules also use different scopes for their members. Members defined within a class are scoped within a specific instance of the class and exist only for the lifetime of the object. To access class members from outside a class, you must use fully qualified names in the format of *Object.Member*.

On the other hand, members declared within a module are publicly accessible by default, and can be accessed by any code that can access the module. This means that variables in a standard module are effectively global variables because they are visible from anywhere in your project, and they exist for the life of the program.

Reusing Classes and Objects

Objects let you declare variables and procedures once and then reuse them whenever needed. For example, if you want to

add a spelling checker to an application you could define all the variables and support functions to provide spell-checking functionality. If you create your spelling checker as a class, you can then reuse it in other applications by adding a reference to the compiled assembly. Better yet, you may be able to save yourself some work by using a spelling checker class that someone else has already developed.

The .NET Framework provides many examples of components that are available for use. The following example uses the TimeZone class in the System namespace. TimeZone provides members that allow you to retrieve information about the time zone of the current computer system.

```
Public Sub examineTimeZone()
    Dim tz As System.TimeZone = System.TimeZone.CurrentTimeZone
    Dim s As String = "Current time zone is "
    s &= CStr(tz.GetUtcOffset(Now).Hours) & " hours and "
    s &= CStr(tz.GetUtcOffset(Now).Minutes) & " minutes "
    s &= "different from UTC (coordinated universal time)"
    s &= vbCrLf & "and is currently "
    If tz.IsDaylightSavingTime(Now) = False Then s &= "not "
    s &= "on ""summer time""."
    MsgBox(s)
End Sub
```

In the preceding example, the first Dim Statement (Visual Basic) declares an object variable of type TimeZone and assigns to it a TimeZone object returned by the CurrentTimeZone property.

Relationships Among Objects

Objects can be related to each other in several ways. The principal kinds of relationship are *hierarchical* and *containment*.

Hierarchical Relationship

When classes are derived from more fundamental classes, they are said to have a *hierarchical relationship*. Class hierarchies are useful when describing items that are a subtype of a more general class.

In the following example, suppose you want to define a special kind of Button that acts like a normal Button but also exposes a method that reverses the foreground and background colors.

To define a class is derived from an already existing class

1. Use a Class Statement (Visual Basic) to define a class from which to create the object you need.

   ```
   Public Class reversibleButton
   ```

   Be sure an **End Class** statement follows the last line of code in your class. By default, the integrated development environment (IDE) automatically generates an **End Class** when you enter a **Class** statement.

2. Follow the **Class** statement immediately with an Inherits Statement. Specify the class from which your new class derives.

   ```
   Inherits System.Windows.Forms.Button
   ```

Your new class inherits all the members defined by the base class.

3. Add the code for the additional members your derived class exposes. For example, you might add a `reverseColors` method, and your derived class might look as follows:

```
Public Class reversibleButton
    Inherits System.Windows.Forms.Button
    Public Sub reverseColors()
        Dim saveColor As System.Drawing.Color = Me.BackColor
        Me.BackColor = Me.ForeColor
        Me.ForeColor = saveColor
    End Sub
End Class
```

If you create an object from the `reversibleButton` class, it can access all the members of the Button class, as well as the `reverseColors` method and any other new members you define on `reversibleButton`.

Derived classes inherit members from the class they are based on, allowing you to add complexity as you progress in a class hierarchy. For more information, see Inheritance Basics (Visual Basic).

Compiling the Code

Be sure the compiler can access the class from which you intend to derive your new class. This might mean fully qualifying its name, as in the preceding example, or identifying its namespace in an Imports Statement (.NET Namespace and Type). If the class is in a different project, you might need to add a reference to that project. For more information, see Managing references in a project.

Containment Relationship

Another way that objects can be related is a *containment relationship*. Container objects logically encapsulate other objects. For example, the OperatingSystem object logically contains a Version object, which it returns through its Version property. Note that the container object does not physically contain any other object.

Collections

One particular type of object containment is represented by *collections*. Collections are groups of similar objects that can be enumerated. Visual Basic supports a specific syntax in the For Each...Next Statement (Visual Basic) that allows you to iterate through the items of a collection. Additionally, collections often allow you to use an Item to retrieve elements by their index or by associating them with a unique string. Collections can be easier to use than arrays because they allow you to add or remove items without using indexes. Because of their ease of use, collections are often used to store forms and controls.

Related Topics

Walkthrough: Defining Classes (Visual Basic)
: Provides a step-by-step description of how to create a class.

Overloaded Properties and Methods (Visual Basic)
: Overloaded Properties and Methods

Inheritance Basics (Visual Basic)
: Covers inheritance modifiers, overriding methods and properties, MyClass, and MyBase.

Object Lifetime: How Objects Are Created and Destroyed (Visual Basic)
: Discusses creating and disposing of class instances.

Anonymous Types (Visual Basic)
: Describes how to create and use anonymous types, which allow you to create objects without writing a class definition for the data type.

Object Initializers: Named and Anonymous Types (Visual Basic)
: Discusses object initializers, which are used to create instances of named and anonymous types by using a single expression.

How to: Infer Property Names and Types in Anonymous Type Declarations (Visual Basic)
: Explains how to infer property names and types in anonymous type declarations. Provides examples of successful and unsuccessful inference.

Walkthrough: Defining Classes (Visual Basic)

Visual Studio 2015

This walkthrough demonstrates how to define classes, which you can then use to create objects. It also shows you how to add properties and methods to the new class, and demonstrates how to initialize an object.

> **Note**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the Visual Studio IDE.

To define a class

1. Create a project by clicking **New Project** on the **File** menu. The **New Project** dialog box appears.
2. Select Windows Application from the list of Visual Basic project templates to display the new project.
3. Add a new class to the project by clicking **Add Class** on the **Project** menu. The **Add New Item** dialog box appears.
4. Select the **Class** template.
5. Name the new class `UserNameInfo.vb`, and then click **Add** to display the code for the new class.

   VB

   ```
   Public Class UserNameInfo
   End Class
   ```

   > **Note**
   >
   > You can use the Visual Basic **Code Editor** to add a class to your startup form by typing the **Class** keyword followed by the name of the new class. The **Code Editor** provides a corresponding **End Class** statement for you.

6. Define a private field for the class by adding the following code between the **Class** and **End Class** statements:

   VB

   ```
   Private userNameValue As String
   ```

   Declaring the field as **Private** means it can be used only within the class. You can make fields available from outside a

class by using access modifiers such as **Public** that provide more access. For more information, see Access Levels in Visual Basic.

7. Define a property for the class by adding the following code:

VB

```
Public Property UserName() As String
    Get
        ' Gets the property value.
        Return userNameValue
    End Get
    Set(ByVal Value As String)
        ' Sets the property value.
        userNameValue = Value
    End Set
End Property
```

8. Define a method for the class by adding the following code:

VB

```
Public Sub Capitalize()
    ' Capitalize the value of the property.
    userNameValue = UCase(userNameValue)
End Sub
```

9. Define a parameterized constructor for the new class by adding a procedure named `Sub New`:

VB

```
Public Sub New(ByVal UserName As String)
    ' Set the property value.
    Me.UserName = UserName
End Sub
```

The `Sub New` constructor is called automatically when an object based on this class is created. This constructor sets the value of the field that holds the user name.

To create a button to test the class

1. Change the startup form to design mode by right-clicking its name in **Solution Explorer** and then clicking **View Designer**. By default, the startup form for Windows Application projects is named Form1.vb. The main form will then appear.

2. Add a button to the main form and double-click it to display the code for the `Button1_Click` event handler. Add the following code to call the test procedure:

VB

```
' Create an instance of the class.
Dim user As New UserNameInfo("Moore, Bobby")
' Capitalize the value of the property.
user.Capitalize()
' Display the value of the property.
MsgBox("The original UserName is: " & user.UserName)
' Change the value of the property.
user.UserName = "Worden, Joe"
' Redisplay the value of the property.
MsgBox("The new UserName is: " & user.UserName)
```

To run your application

1. Run your application by pressing F5. Click the button on the form to call the test procedure. It displays a message stating that the original `UserName` is "MOORE, BOBBY", because the procedure called the `Capitalize` method of the object.

2. Click **OK** to dismiss the message box. The `Button1 Click` procedure changes the value of the `UserName` property and displays a message stating that the new value of `UserName` is "Worden, Joe".

See Also

Object-Oriented Programming (C# and Visual Basic)
Objects and Classes in Visual Basic

Overloaded Properties and Methods (Visual Basic)

Visual Studio 2015

Overloading is the creation of more than one procedure, instance constructor, or property in a class with the same name but different argument types.

Overloading Usage

Overloading is especially useful when your object model dictates that you employ identical names for procedures that operate on different data types. For example, a class that can display several different data types could have `Display` procedures that look like this:

VB

```
Overloads Sub Display(ByVal theChar As Char)
    ' Add code that displays Char data.
End Sub
Overloads Sub Display(ByVal theInteger As Integer)
    ' Add code that displays Integer data.
End Sub
Overloads Sub Display(ByVal theDouble As Double)
    ' Add code that displays Double data.
End Sub
```

Without overloading, you would need to create distinct names for each procedure, even though they do the same thing, as shown next:

VB

```
Sub DisplayChar(ByVal theChar As Char)
    ' Add code that displays Char data.
End Sub
Sub DisplayInt(ByVal theInteger As Integer)
    ' Add code that displays Integer data.
End Sub
Sub DisplayDouble(ByVal theDouble As Double)
    ' Add code that displays Double data.
End Sub
```

Overloading makes it easier to use properties or methods because it provides a choice of data types that can be used. For example, the overloaded `Display` method discussed previously can be called with any of the following lines of code:

VB

```
' Call Display with a literal of type Char.
Display("9"c)
' Call Display with a literal of type Integer.
Display(9)
' Call Display with a literal of type Double.
Display(9.9R)
```

At run time, Visual Basic calls the correct procedure based on the data types of the parameters you specify.

Overloading Rules

You create an overloaded member for a class by adding two or more properties or methods with the same name. Except for overloaded derived members, each overloaded member must have different parameter lists, and the following items cannot be used as a differentiating feature when overloading a property or procedure:

- Modifiers, such as **ByVal** or **ByRef**, that apply to a member, or parameters of the member.
- Names of parameters
- Return types of procedures

The **Overloads** keyword is optional when overloading, but if any overloaded member uses the **Overloads** keyword, then all other overloaded members with the same name must also specify this keyword.

Derived classes can overload inherited members with members that have identical parameters and parameter types, a process known as *shadowing by name and signature*. If the **Overloads** keyword is used when shadowing by name and signature, the derived class's implementation of the member will be used instead of the implementation in the base class, and all other overloads for that member will be available to instances of the derived class.

If the **Overloads** keyword is omitted when overloading an inherited member with a member that has identical parameters and parameter types, then the overloading is called *shadowing by name*. Shadowing by name replaces the inherited implementation of a member, and it makes all other overloads unavailable to instances of the derived class and its decedents.

The **Overloads** and **Shadows** modifiers cannot both be used with the same property or method.

Example

The following example creates overloaded methods that accept either a **String** or **Decimal** representation of a dollar amount and return a string containing the sales tax.

To use this example to create an overloaded method

1. Open a new project and add a class named `TaxClass`.

2. Add the following code to the `TaxClass` class.

VB

```
Public Class TaxClass
    Overloads Function TaxAmount(ByVal decPrice As Decimal,
         ByVal TaxRate As Single) As String
        TaxAmount = "Price is a Decimal. Tax is $" &
           (CStr(decPrice * TaxRate))
    End Function

    Overloads Function TaxAmount(ByVal strPrice As String,
          ByVal TaxRate As Single) As String
        TaxAmount = "Price is a String. Tax is $" &
           CStr((CDec(strPrice) * TaxRate))
    End Function
End Class
```

3. Add the following procedure to your form.

VB

```
Sub ShowTax()
    ' 8% tax rate.
    Const TaxRate As Single = 0.08
    ' $64.00 Purchase as a String.
    Dim strPrice As String = "64.00"
    ' $64.00 Purchase as a Decimal.
    Dim decPrice As Decimal = 64
    Dim aclass As New TaxClass
    'Call the same method with two different kinds of data.
    MsgBox(aclass.TaxAmount(strPrice, TaxRate))
    MsgBox(aclass.TaxAmount(decPrice, TaxRate))
End Sub
```

4. Add a button to your form and call the `ShowTax` procedure from the `Button1_Click` event of the button.

5. Run the project and click the button on the form to test the overloaded `ShowTax` procedure.

At run time, the compiler chooses the appropriate overloaded function that matches the parameters being used. When you click the button, the overloaded method is called first with a `Price` parameter that is a string and the message, "Price is a String. Tax is $5.12" is displayed. `TaxAmount` is called with a **Decimal** value the second time and the message, "Price is a Decimal. Tax is $5.12" is displayed.

See Also

Objects and Classes in Visual Basic
Shadowing in Visual Basic
Sub Statement (Visual Basic)
Inheritance Basics (Visual Basic)
Shadows (Visual Basic)

ByVal (Visual Basic)
ByRef (Visual Basic)
Overloads (Visual Basic)
Shadows (Visual Basic)

Inheritance Basics (Visual Basic)

Visual Studio 2015

The **Inherits** statement is used to declare a new class, called a *derived class*, based on an existing class, known as a *base class*. Derived classes inherit, and can extend, the properties, methods, events, fields, and constants defined in the base class. The following section describes some of the rules for inheritance, and the modifiers you can use to change the way classes inherit or are inherited:

- By default, all classes are inheritable unless marked with the **NotInheritable** keyword. Classes can inherit from other classes in your project or from classes in other assemblies that your project references.
- Unlike languages that allow multiple inheritance, Visual Basic allows only single inheritance in classes; that is, derived classes can have only one base class. Although multiple inheritance is not allowed in classes, classes can implement multiple interfaces, which can effectively accomplish the same ends.
- To prevent exposing restricted items in a base class, the access type of a derived class must be equal to or more restrictive than its base class. For example, a **Public** class cannot inherit a **Friend** or a **Private** class, and a **Friend** class cannot inherit a **Private** class.

Inheritance Modifiers

Visual Basic introduces the following class-level statements and modifiers to support inheritance:

- **Inherits** statement — Specifies the base class.
- **NotInheritable** modifier — Prevents programmers from using the class as a base class.
- **MustInherit** modifier — Specifies that the class is intended for use as a base class only. Instances of **MustInherit** classes cannot be created directly; they can only be created as base class instances of a derived class. (Other programming languages, such as C++ and C#, use the term *abstract class* to describe such a class.)

Overriding Properties and Methods in Derived Classes

By default, a derived class inherits properties and methods from its base class. If an inherited property or method has to behave differently in the derived class it can be *overridden*. That is, you can define a new implementation of the method in the derived class. The following modifiers are used to control how properties and methods are overridden:

- **Overridable** — Allows a property or method in a class to be overridden in a derived class.
- **Overrides** — Overrides an **Overridable** property or method defined in the base class.
- **NotOverridable** — Prevents a property or method from being overridden in an inheriting class. By default, **Public**

methods are **NotOverridable**.

- **MustOverride** — Requires that a derived class override the property or method. When the **MustOverride** keyword is used, the method definition consists of just the **Sub**, **Function**, or **Property** statement. No other statements are allowed, and specifically there is no **End Sub** or **End Function** statement. **MustOverride** methods must be declared in **MustInherit** classes.

Suppose you want to define classes to handle payroll. You could define a generic `Payroll` class that contains a `RunPayroll` method that calculates payroll for a typical week. You could then use `Payroll` as a base class for a more specialized `BonusPayroll` class, which could be used when distributing employee bonuses.

The `BonusPayroll` class can inherit, and override, the `PayEmployee` method defined in the base `Payroll` class.

The following example defines a base class, `Payroll`, and a derived class, `BonusPayroll`, which overrides an inherited method, `PayEmployee`. A procedure, `RunPayroll`, creates and then passes a `Payroll` object and a `BonusPayroll` object to a function, `Pay`, that executes the `PayEmployee` method of both objects.

VB

```
Const BonusRate As Decimal = 1.45D
Const PayRate As Decimal = 14.75D

Class Payroll
    Overridable Function PayEmployee(
        ByVal HoursWorked As Decimal,
        ByVal PayRate As Decimal) As Decimal

        PayEmployee = HoursWorked * PayRate
    End Function
End Class

Class BonusPayroll
    Inherits Payroll
    Overrides Function PayEmployee(
        ByVal HoursWorked As Decimal,
        ByVal PayRate As Decimal) As Decimal

        ' The following code calls the original method in the base
        ' class, and then modifies the returned value.
        PayEmployee = MyBase.PayEmployee(HoursWorked, PayRate) * BonusRate
    End Function
End Class

Sub RunPayroll()
    Dim PayrollItem As Payroll = New Payroll
    Dim BonusPayrollItem As New BonusPayroll
    Dim HoursWorked As Decimal = 40

    MsgBox("Normal pay is: " &
        PayrollItem.PayEmployee(HoursWorked, PayRate))
    MsgBox("Pay with bonus is: " &
        BonusPayrollItem.PayEmployee(HoursWorked, PayRate))
End Sub
```

The MyBase Keyword

The **MyBase** keyword behaves like an object variable that refers to the base class of the current instance of a class. **MyBase** is frequently used to access base class members that are overridden or shadowed in a derived class. In particular, `MyBase.New` is used to explicitly call a base class constructor from a derived class constructor.

For example, suppose you are designing a derived class that overrides a method inherited from the base class. The overridden method can call the method in the base class and modify the return value as shown in the following code fragment:

VB

```
Class DerivedClass
    Inherits BaseClass
    Public Overrides Function CalculateShipping(
```

```
        ByVal Dist As Double,
        ByVal Rate As Double) As Double

        ' Call the method in the base class and modify the return value.
        Return MyBase.CalculateShipping(Dist, Rate) * 2
    End Function
End Class
```

The following list describes restrictions on using **MyBase**:

- **MyBase** refers to the immediate base class and its inherited members. It cannot be used to access **Private** members in the class.
- **MyBase** is a keyword, not a real object. **MyBase** cannot be assigned to a variable, passed to procedures, or used in an **Is** comparison.
- The method that **MyBase** qualifies does not have to be defined in the immediate base class; it may instead be defined in an indirectly inherited base class. In order for a reference qualified by **MyBase** to compile correctly, some base class must contain a method matching the name and types of parameters that appear in the call.
- You cannot use **MyBase** to call **MustOverride** base class methods.
- **MyBase** cannot be used to qualify itself. Therefore, the following code is not valid:

  ```
  MyBase.MyBase.BtnOK_Click()
  ```

- **MyBase** cannot be used in modules.
- **MyBase** cannot be used to access base class members that are marked as **Friend** if the base class is in a different assembly.

For more information and another example, see How to: Access a Variable Hidden by a Derived Class (Visual Basic).

The MyClass Keyword

The **MyClass** keyword behaves like an object variable that refers to the current instance of a class as originally implemented. **MyClass** resembles **Me**, but every method and property call on **MyClass** is treated as if the method or property were NotOverridable (Visual Basic). Therefore, the method or property is not affected by overriding in a derived class.

- **MyClass** is a keyword, not a real object. **MyClass** cannot be assigned to a variable, passed to procedures, or used in an **Is** comparison.
- **MyClass** refers to the containing class and its inherited members.
- **MyClass** can be used as a qualifier for **Shared** members.
- **MyClass** cannot be used inside a **Shared** method, but can be used inside an instance method to access a shared member of a class.

- **MyClass** cannot be used in standard modules.
- **MyClass** can be used to qualify a method that is defined in a base class and that has no implementation of the method provided in that class. Such a reference has the same meaning as **MyBase.***Method*.

The following example compares **Me** and **MyClass**.

```
Class baseClass
    Public Overridable Sub testMethod()
        MsgBox("Base class string")
    End Sub
    Public Sub useMe()
        ' The following call uses the calling class's method, even if
        ' that method is an override.
        Me.testMethod()
    End Sub
    Public Sub useMyClass()
        ' The following call uses this instance's method and not any
        ' override.
        MyClass.testMethod()
    End Sub
End Class
Class derivedClass : Inherits baseClass
    Public Overrides Sub testMethod()
        MsgBox("Derived class string")
    End Sub
End Class
Class testClasses
    Sub startHere()
        Dim testObj As derivedClass = New derivedClass()
        ' The following call displays "Derived class string".
        testObj.useMe()
        ' The following call displays "Base class string".
        testObj.useMyClass()
    End Sub
End Class
```

Even though `derivedClass` overrides `testMethod`, the **MyClass** keyword in `useMyClass` nullifies the effects of overriding, and the compiler resolves the call to the base class version of `testMethod`.

See Also

Inherits Statement
Me, My, MyBase, and MyClass in Visual Basic

Object Lifetime: How Objects Are Created and Destroyed (Visual Basic)

Visual Studio 2015

An instance of a class, an object, is created by using the **New** keyword. Initialization tasks often must be performed on new objects before they are used. Common initialization tasks include opening files, connecting to databases, and reading values of registry keys. Visual Basic controls the initialization of new objects using procedures called *constructors* (special methods that allow control over initialization).

After an object leaves scope, it is released by the common language runtime (CLR). Visual Basic controls the release of system resources using procedures called *destructors*. Together, constructors and destructors support the creation of robust and predictable class libraries.

Using Constructors and Destructors

Constructors and destructors control the creation and destruction of objects. The **Sub New** and **Sub Finalize** procedures in Visual Basic initialize and destroy objects; they replace the **Class_Initialize** and **Class_Terminate** methods used in Visual Basic 6.0 and earlier versions.

Sub New

The **Sub New** constructor can run only once when a class is created. It cannot be called explicitly anywhere other than in the first line of code of another constructor from either the same class or from a derived class. Furthermore, the code in the **Sub New** method always runs before any other code in a class. Visual Basic 2005 and later versions implicitly create a **Sub New** constructor at run time if you do not explicitly define a **Sub New** procedure for a class.

To create a constructor for a class, create a procedure named **Sub New** anywhere in the class definition. To create a parameterized constructor, specify the names and data types of arguments to **Sub New** just as you would specify arguments for any other procedure, as in the following code:

VB

```
Sub New(ByVal s As String)
```

Constructors are frequently overloaded, as in the following code:

VB

```
Sub New(ByVal s As String, i As Integer)
```

When you define a class derived from another class, the first line of a constructor must be a call to the constructor of the base class, unless the base class has an accessible constructor that takes no parameters. A call to the base class that contains the above constructor, for example, would be `MyBase.New(s)`. Otherwise, `MyBase.New` is optional, and the Visual Basic runtime calls it implicitly.

After you write the code to call the parent object's constructor, you can add any additional initialization code to the **Sub New** procedure. **Sub New** can accept arguments when called as a parameterized constructor. These parameters are passed from the procedure calling the constructor, for example, `Dim AnObject As New ThisClass(X)`.

Sub Finalize

Before releasing objects, the CLR automatically calls the **Finalize** method for objects that define a **Sub Finalize** procedure. The **Finalize** method can contain code that needs to execute just before an object is destroyed, such as code for closing files and saving state information. There is a slight performance penalty for executing **Sub Finalize**, so you should define a **Sub Finalize** method only when you need to release objects explicitly.

> **Note**
>
> The garbage collector in the CLR does not (and cannot) dispose of *unmanaged objects*, objects that the operating system executes directly, outside the CLR environment. This is because different unmanaged objects must be disposed of in different ways. That information is not directly associated with the unmanaged object; it must be found in the documentation for the object. A class that uses unmanaged objects must dispose of them in its **Finalize** method.

The **Finalize** destructor is a protected method that can be called only from the class it belongs to, or from derived classes. The system calls **Finalize** automatically when an object is destroyed, so you should not explicitly call **Finalize** from outside of a derived class's **Finalize** implementation.

Unlike **Class_Terminate**, which executes as soon as an object is set to nothing, there is usually a delay between when an object loses scope and when Visual Basic calls the **Finalize** destructor. Visual Basic 2005 and later versions allow for a second kind of destructor, Dispose, which can be explicitly called at any time to immediately release resources.

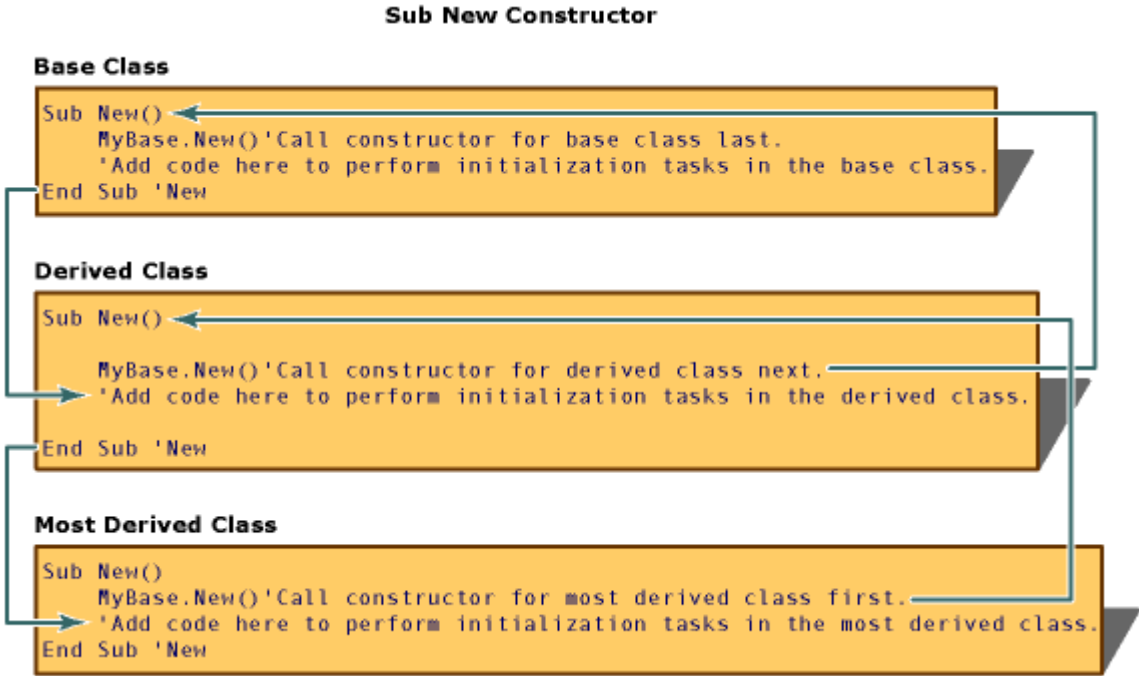
> **Note**
>
> A **Finalize** destructor should not throw exceptions, because they cannot be handled by the application and can cause the application to terminate.

How New and Finalize Methods Work in a Class Hierarchy

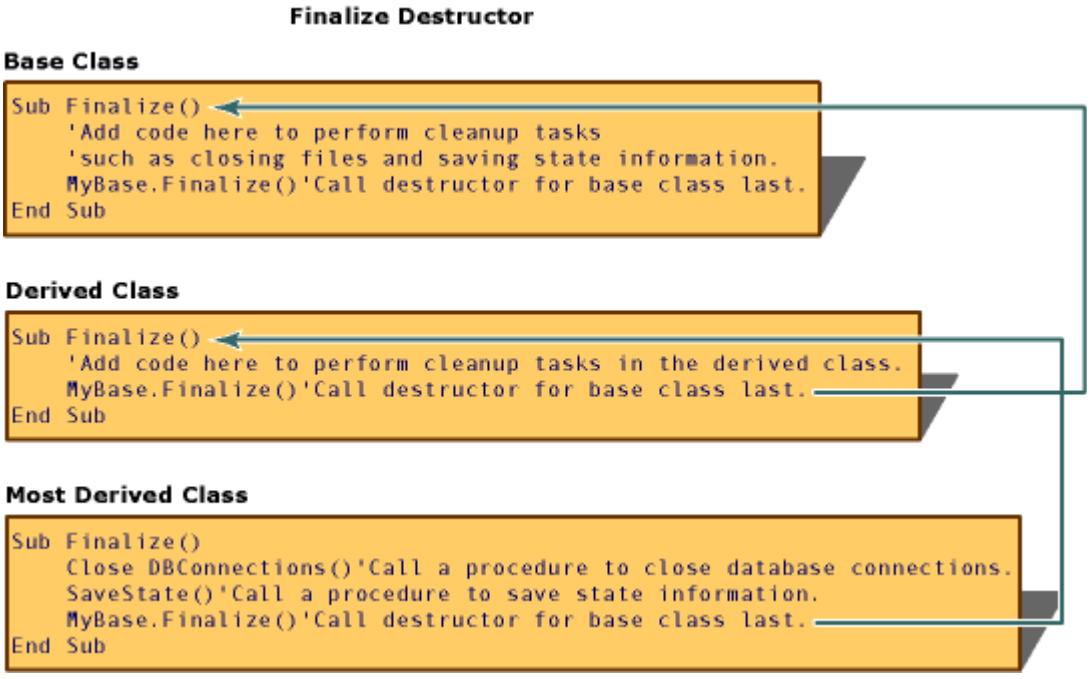
Whenever an instance of a class is created, the common language runtime (CLR) attempts to execute a procedure named **New**, if it exists in that object. **New** is a type of procedure called a *constructor* that is used to initialize new objects before any other code in an object executes. A **New** constructor can be used to open files, connect to databases, initialize variables, and take care of any other tasks that need to be done before an object can be used.

When an instance of a derived class is created, the **Sub New** constructor of the base class executes first, followed by constructors in derived classes. This happens because the first line of code in a **Sub New** constructor uses the syntax `MyBase.New()`to call the constructor of the class immediately above itself in the class hierarchy. The **Sub New** constructor is then called for each class in the class hierarchy until the constructor for the base class is reached. At that point, the code in the constructor for the base class executes, followed by the code in each constructor in all derived

classes and the code in the most derived classes is executed last.



When an object is no longer needed, the CLR calls the Finalize method for that object before freeing its memory. The Finalize method is called a *destructor* because it performs cleanup tasks, such as saving state information, closing files and connections to databases, and other tasks that must be done before releasing the object.



IDisposable Interface

Class instances often control resources not managed by the CLR, such as Windows handles and database connections. These resources must be disposed of in the **Finalize** method of the class, so that they will be released when the object is destroyed by the garbage collector. However, the garbage collector destroys objects only when the CLR requires more

free memory. This means that the resources may not be released until long after the object goes out of scope.

To supplement garbage collection, your classes can provide a mechanism to actively manage system resources if they implement the IDisposable interface. IDisposable has one method, Dispose, which clients should call when they finish using an object. You can use the Dispose method to immediately release resources and perform tasks such as closing files and database connections. Unlike the **Finalize** destructor, the Dispose method is not called automatically. Clients of a class must explicitly call Dispose when you want to immediately release resources.

Implementing IDisposable

A class that implements the IDisposable interface should include these sections of code:

- A field for keeping track of whether the object has been disposed:

```
Protected disposed As Boolean = False
```

- An overload of the Dispose that frees the class's resources. This method should be called by the Dispose and **Finalize** methods of the base class:

```
Protected Overridable Sub Dispose(ByVal disposing As Boolean)
    If Not Me.disposed Then
        If disposing Then
            ' Insert code to free managed resources.
        End If
        ' Insert code to free unmanaged resources.
    End If
    Me.disposed = True
End Sub
```

- An implementation of Dispose that contains only the following code:

```
Public Sub Dispose() Implements IDisposable.Dispose
    Dispose(True)
    GC.SuppressFinalize(Me)
End Sub
```

- An override of the **Finalize** method that contains only the following code:

```
Protected Overrides Sub Finalize()
    Dispose(False)
    MyBase.Finalize()
End Sub
```

Deriving from a Class that Implements IDisposable

A class that derives from a base class that implements the IDisposable interface does not need to override any of the base methods unless it uses additional resources that need to be disposed. In that situation, the derived class should override the base class's `Dispose(disposing)` method to dispose of the derived class's resources. This override must call the base class's `Dispose(disposing)` method.

```
Protected Overrides Sub Dispose(ByVal disposing As Boolean)
    If Not Me.disposed Then
        If disposing Then
            ' Insert code to free managed resources.
        End If
        ' Insert code to free unmanaged resources.
    End If
    MyBase.Dispose(disposing)
End Sub
```

A derived class should not override the base class's Dispose and **Finalize** methods. When those methods are called from an instance of the derived class, the base class's implementation of those methods call the derived class's override of the `Dispose(disposing)` method.

Garbage Collection and the Finalize Destructor

The .NET Framework uses the *reference-tracing garbage collection* system to periodically release unused resources. Visual Basic 6.0 and earlier versions used a different system called *reference counting* to manage resources. Although both systems perform the same function automatically, there are a few important differences.

The CLR periodically destroys objects when the system determines that such objects are no longer needed. Objects are released more quickly when system resources are in short supply, and less frequently otherwise. The delay between when an object loses scope and when the CLR releases it means that, unlike with objects in Visual Basic 6.0 and earlier versions, you cannot determine exactly when the object will be destroyed. In such a situation, objects are said to have *non-deterministic lifetime*. In most cases, non-deterministic lifetime does not change how you write applications, as long as you remember that the **Finalize** destructor may not immediately execute when an object loses scope.

Another difference between the garbage-collection systems involves the use of **Nothing**. To take advantage of reference counting in Visual Basic 6.0 and earlier versions, programmers sometimes assigned **Nothing** to object variables to release the references those variables held. If the variable held the last reference to the object, the object's resources were released immediately. In later versions of Visual Basic, while there may be cases in which this procedure is still valuable, performing it never causes the referenced object to release its resources immediately. To release resources immediately, use the object's Dispose method, if available. The only time you should set a variable to **Nothing** is when its lifetime is long relative to the time the garbage collector takes to detect orphaned objects.

See Also

Dispose
Initialization and Termination of Components
New Operator (Visual Basic)
Cleaning Up Unmanaged Resources
Nothing (Visual Basic)

Object Initializers: Named and Anonymous Types (Visual Basic)

Visual Studio 2015

Object initializers enable you to specify properties for a complex object by using a single expression. They can be used to create instances of named types and of anonymous types.

Declarations

Declarations of instances of named and anonymous types can look almost identical, but their effects are not the same. Each category has abilities and restrictions of its own. The following example shows a convenient way to declare and initialize an instance of a named class, `Customer`, by using an object initializer list. Notice that the name of the class is specified after the keyword **New**.

VB

```
Dim namedCust = New Customer With {.Name = "Terry Adams"}
```

An anonymous type has no usable name. Therefore an instantiation of an anonymous type cannot include a class name.

VB

```
Dim anonymousCust = New With {.Name = "Hugo Garcia"}
```

The requirements and results of the two declarations are not the same. For `namedCust`, a `Customer` class that has a `Name` property must already exist, and the declaration creates an instance of that class. For `anonymousCust`, the compiler defines a new class that has one property, a string called `Name`, and creates a new instance of that class.

Named Types

Object initializers provide a simple way to call the constructor of a type and then set the values of some or all properties in a single statement. The compiler invokes the appropriate constructor for the statement: the default constructor if no arguments are presented, or a parameterized constructor if one or more arguments are sent. After that, the specified properties are initialized in the order in which they are presented in the initializer list.

Each initialization in the initializer list consists of the assignment of an initial value to a member of the class. The names and data types of the members are determined when the class is defined. In the following examples, the `Customer` class must exist, and must have members named `Name` and `City` that can accept string values.

VB

```
Dim cust0 As Customer = New Customer With {.Name = "Toni Poe",
```

```
                                                .City = "Louisville"}
```

Alternatively, you can obtain the same result by using the following code:

VB

```
Dim cust1 As New Customer With {.Name = "Toni Poe",
                                .City = "Louisville"}
```

Each of these declarations is equivalent to the following example, which creates a `Customer` object by using the default constructor, and then specifies initial values for the `Name` and `City` properties by using a **With** statement.

VB

```
Dim cust2 As New Customer()
With cust2
    .Name = "Toni Poe"
    .City = "Louisville"
End With
```

If the `Customer` class contains a parameterized constructor that enables you to send in a value for `Name`, for example, you can also declare and initialize a `Customer` object in the following ways:

VB

```
Dim cust3 As Customer =
    New Customer("Toni Poe") With {.City = "Louisville"}
' --or--
Dim cust4 As New Customer("Toni Poe") With {.City = "Louisville"}
```

You do not have to initialize all properties, as the following code shows.

VB

```
Dim cust5 As Customer = New Customer With {.Name = "Toni Poe"}
```

However, the initialization list cannot be empty. Uninitialized properties retain their default values.

Type Inference with Named Types

You can shorten the code for the declaration of `cust1` by combining object initializers and local type inference. This enables you to omit the **As** clause in the variable declaration. The data type of the variable is inferred from the type of the object that is created by the assignment. In the following example, the type of `cust6` is `Customer`.

VB

```
Dim cust6 = New Customer With {.Name = "Toni Poe",
                               .City = "Louisville"}
```

Remarks About Named Types

- A class member cannot be initialized more than one time in the object initializer list. The declaration of `cust7` causes an error.

  VB

  ```
  '' This code does not compile because Name is initialized twice.
  ' Dim cust7 = New Customer With {.Name = "Toni Poe",
  '                                .City = "Louisville",
  '                                .Name = "Blue Yonder Airlines"}
  ```

- A member can be used to initialize itself or another field. If a member is accessed before it has been initialized, as in the following declaration for `cust8`, the default value will be used. Remember that when a declaration that uses an object initializer is processed, the first thing that happens is that the appropriate constructor is invoked. After that, the individual fields in the initializer list are initialized. In the following examples, the default value for `Name` is assigned for `cust8`, and an initialized value is assigned in `cust9`.

  VB

  ```
  Dim cust8 = New Customer With {.Name = .Name & ", President"}
  Dim cust9 = New Customer With {.Name = "Toni Poe",
                                 .Title = .Name & ", President"}
  ```

  The following example uses the parameterized constructor from `cust3` and `cust4` to declare and initialize `cust10` and `cust11`.

  VB

  ```
  Dim cust10 = New Customer("Toni Poe") With {.Name = .Name & ", President"}
  ' --or--
  Dim cust11 As New Customer("Toni Poe") With {.Name = .Name & ", President"}
  ```

- Object initializers can be nested. In the following example, `AddressClass` is a class that has two properties, `City` and `State`, and the `Customer` class has an `Address` property that is an instance of `AddressClass`.

  VB

  ```
  Dim cust12 =
  ```

```
    New Customer With {.Name = "Toni Poe",
                       .Address =
                           New AddressClass With {.City = "Louisville",
                                                  .State = "Kentucky"}}
Console.WriteLine(cust12.Address.State)
```

- The initialization list cannot be empty.
- The instance being initialized cannot be of type Object.
- Class members being initialized cannot be shared members, read-only members, constants, or method calls.
- Class members being initialized cannot be indexed or qualified. The following examples raise compiler errors:

```
'' Not valid.

' Dim c1 = New Customer With {.OrderNumbers(0) = 148662}

' Dim c2 = New Customer with {.Address.City = "Springfield"}
```

Anonymous Types

Anonymous types use object initializers to create instances of new types that you do not explicitly define and name. Instead, the compiler generates a type according to the properties you designate in the object initializer list. Because the name of the type is not specified, it is referred to as an *anonymous type*. For example, compare the following declaration to the earlier one for `cust6`.

VB

```
Dim cust13 = New With {.Name = "Toni Poe",
                       .City = "Louisville"}
```

The only difference syntactically is that no name is specified after **New** for the data type. However, what happens is quite different. The compiler defines a new anonymous type that has two properties, `Name` and `City`, and creates an instance of it with the specified values. Type inference determines the types of `Name` and `City` in the example to be strings.

Caution

The name of the anonymous type is generated by the compiler, and may vary from compilation to compilation. Your code should not use or rely on the name of an anonymous type.

Because the name of the type is not available, you cannot use an **As** clause to declare `cust13`. Its type must be inferred. Without using late binding, this limits the use of anonymous types to local variables.

Anonymous types provide critical support for LINQ queries. For more information about the use of anonymous types in

queries, see Anonymous Types (Visual Basic) and Introduction to LINQ in Visual Basic.

Remarks About Anonymous Types

- Typically, all or most of the properties in an anonymous type declaration will be key properties, which are indicated by typing the keyword **Key** in front of the property name.

  VB

  ```
  Dim anonymousCust1 = New With {Key .Name = "Hugo Garcia",
                                 Key .City = "Louisville"}
  ```

  For more information about key properties, see Key (Visual Basic).

- Like named types, initializer lists for anonymous type definitions must declare at least one property.

  VB

  ```
  Dim anonymousCust = New With {.Name = "Hugo Garcia"}
  ```

- When an instance of an anonymous type is declared, the compiler generates a matching anonymous type definition. The names and data types of the properties are taken from the instance declaration, and are included by the compiler in the definition. The properties are not named and defined in advance, as they would be for a named type. Their types are inferred. You cannot specify the data types of the properties by using an **As** clause.

- Anonymous types can also establish the names and values of their properties in several other ways. For example, an anonymous type property can take both the name and the value of a variable, or the name and value of a property of another object.

  VB

  ```
  ' Create a variable, Name, and give it an initial value.
  Dim Name = "Hugo Garcia"

  ' Variable anonymousCust2 will have one property, Name, with
  ' "Hugo Garcia" as its initial value.
  Dim anonymousCust2 = New With {Key Name}

  ' The next declaration uses a property from namedCust, defined
  ' in an earlier example. After the declaration, anonymousCust3 will
  ' have one property, Name, with "Terry Adams" as its value.
  Dim anonymousCust3 = New With {Key namedCust.Name}
  ```

  For more information about the options for defining properties in anonymous types, see How to: Infer Property Names and Types in Anonymous Type Declarations (Visual Basic).

See Also

Local Type Inference (Visual Basic)
Anonymous Types (Visual Basic)
Introduction to LINQ in Visual Basic
How to: Infer Property Names and Types in Anonymous Type Declarations (Visual Basic)
Key (Visual Basic)
How to: Declare an Object by Using an Object Initializer (Visual Basic)

How to: Declare an Object by Using an Object Initializer (Visual Basic)

Visual Studio 2015

Object initializers enable you to declare and instantiate an instance of a class in a single statement. In addition, you can initialize one or more members of the instance at the same time, without invoking a parameterized constructor.

When you use an object initializer to create an instance of a named type, the default constructor for the class is called, followed by initialization of designated members in the order you specify.

The following procedure shows how to create an instance of a `Student` class in three different ways. The class has first name, last name, and class year properties, among others. Each of the three declarations creates a new instance of `Student`, with property `First` set to "Michael", property `Last` set to "Tucker", and all other members set to their default values. The result of each declaration in the procedure is equivalent to the following example, which does not use an object initializer.

VB

```
Dim student0 As New Student
With student0
    .First = "Michael"
    .Last = "Tucker"
End With
```

For an implementation of the `Student` class, see How to: Create a List of Items. You can copy the code from that topic to set up the class and create a list of `Student` objects to work with.

To create an object of a named class by using an object initializer

1. Begin the declaration as if you planned to use a constructor.

   ```
   Dim student1 As New Student
   ```

2. Type the keyword **With**, followed by an initialization list in braces.

   ```
   Dim student1 As New Student With { <initialization list> }
   ```

3. In the initialization list, include each property that you want to initialize and assign an initial value to it. The name of the property is preceded by a period.

   VB

   ```
   Dim student1 As New Student With {.First = "Michael",
                                     .Last = "Tucker"}
   ```

You can initialize one or more members of the class.

4. Alternatively, you can declare a new instance of the class and then assign a value to it. First, declare an instance of `Student`:

   ```
   Dim student2 As Student
   ```

5. Begin the creation of an instance of `Student` in the normal way.

   ```
   Dim student2 As Student = New Student
   ```

6. Type **With** and then an object initializer to initialize one or more members of the new instance.

   VB

   ```
   Dim student2 As Student = New Student With {.First = "Michael",
                                               .Last = "Tucker"}
   ```

7. You can simplify the definition in the previous step by omitting `As Student`. If you do this, the compiler determines that `student3` is an instance of `Student` by using local type inference.

   VB

   ```
   Dim student3 = New Student With {.First = "Michael",
                                    .Last = "Tucker"}
   ```

   For more information, see Local Type Inference (Visual Basic).

See Also

Local Type Inference (Visual Basic)
How to: Create a List of Items
Object Initializers: Named and Anonymous Types (Visual Basic)
Anonymous Types (Visual Basic)

Anonymous Types (Visual Basic)

Visual Studio 2015

Visual Basic supports anonymous types, which enable you to create objects without writing a class definition for the data type. Instead, the compiler generates a class for you. The class has no usable name, inherits directly from Object, and contains the properties you specify in declaring the object. Because the name of the data type is not specified, it is referred to as an *anonymous type*.

The following example declares and creates variable `product` as an instance of an anonymous type that has two properties, `Name` and `Price`.

VB

```
' Variable product is an instance of a simple anonymous type.
Dim product = New With {Key .Name = "paperclips", .Price = 1.29}
```

A *query expression* uses anonymous types to combine columns of data selected by a query. You cannot define the type of the result in advance, because you cannot predict the columns a particular query might select. Anonymous types enable you to write a query that selects any number of columns, in any order. The compiler creates a data type that matches the specified properties and the specified order.

In the following examples, `products` is a list of product objects, each of which has many properties. Variable `namePriceQuery` holds the definition of a query that, when it is executed, returns a collection of instances of an anonymous type that has two properties, `Name` and `Price`.

VB

```
Dim namePriceQuery = From prod In products
                     Select prod.Name, prod.Price
```

Variable `nameQuantityQuery` holds the definition of a query that, when it is executed, returns a collection of instances of an anonymous type that has two properties, `Name` and `OnHand`.

VB

```
Dim nameQuantityQuery = From prod In products
                        Select prod.Name, prod.OnHand
```

For more information about the code created by the compiler for an anonymous type, see Anonymous Type Definition (Visual Basic).

Caution

The name of the anonymous type is compiler generated and may vary from compilation to compilation. Your code should not use or rely on the name of an anonymous type because the name might change when a project is recompiled.

Declaring an Anonymous Type

The declaration of an instance of an anonymous type uses an initializer list to specify the properties of the type. You can specify only properties when you declare an anonymous type, not other class elements such as methods or events. In the following example, `product1` is an instance of an anonymous type that has two properties: `Name` and `Price`.

VB

```
' Variable product1 is an instance of a simple anonymous type.
Dim product1 = New With {.Name = "paperclips", .Price = 1.29}
' -or-
' product2 is an instance of an anonymous type with key properties.
Dim product2 = New With {Key .Name = "paperclips", Key .Price = 1.29}
```

If you designate properties as key properties, you can use them to compare two anonymous type instances for equality. However, the values of key properties cannot be changed. See the Key Properties section later in this topic for more information.

Notice that declaring an instance of an anonymous type is like declaring an instance of a named type by using an object initializer:

VB

```
' Variable product3 is an instance of a class named Product.
Dim product3 = New Product With {.Name = "paperclips", .Price = 1.29}
```

For more information about other ways to specify anonymous type properties, see How to: Infer Property Names and Types in Anonymous Type Declarations (Visual Basic).

Key Properties

Key properties differ from non-key properties in several fundamental ways:

- Only the values of key properties are compared in order to determine whether two instances are equal.
- The values of key properties are read-only and cannot be changed.
- Only key property values are included in the compiler-generated hash code algorithm for an anonymous type.

Equality

Instances of anonymous types can be equal only if they are instances of the same anonymous type. The compiler treats

two instances as instances of the same type if they meet the following conditions:

- They are declared in the same assembly.
- Their properties have the same names, the same inferred types, and are declared in the same order. Name comparisons are not case-sensitive.
- The same properties in each are marked as key properties.
- At least one property in each declaration is a key property.

An instance of an anonymous types that has no key properties is equal only to itself.

VB

```
' prod1 and prod2 have no key values.
Dim prod1 = New With {.Name = "paperclips", .Price = 1.29}
Dim prod2 = New With {.Name = "paperclips", .Price = 1.29}

' The following line displays False, because prod1 and prod2 have no
' key properties.
Console.WriteLine(prod1.Equals(prod2))

' The following statement displays True because prod1 is equal to itself.
Console.WriteLine(prod1.Equals(prod1))
```

Two instances of the same anonymous type are equal if the values of their key properties are equal. The following examples illustrate how equality is tested.

VB

```
Dim prod3 = New With {Key .Name = "paperclips", Key .Price = 1.29}
Dim prod4 = New With {Key .Name = "paperclips", Key .Price = 1.29}
' The following line displays True, because prod3 and prod4 are
' instances of the same anonymous type, and the values of their
' key properties are equal.
Console.WriteLine(prod3.Equals(prod4))

Dim prod5 = New With {Key .Name = "paperclips", Key .Price = 1.29}
Dim prod6 = New With {Key .Name = "paperclips", Key .Price = 1.29,
                      .OnHand = 423}
' The following line displays False, because prod5 and prod6 do not
' have the same properties.
Console.WriteLine(prod5.Equals(prod6))

Dim prod7 = New With {Key .Name = "paperclips", Key .Price = 1.29,
                      .OnHand = 24}
Dim prod8 = New With {Key .Name = "paperclips", Key .Price = 1.29,
                      .OnHand = 423}
' The following line displays True, because prod7 and prod8 are
' instances of the same anonymous type, and the values of their
' key properties are equal. The equality check does not compare the
```

```
' values of the non-key field.
Console.WriteLine(prod7.Equals(prod8))
```

Read-Only Values

The values of key properties cannot be changed. For example, in `prod8` in the previous example, the `Name` and `Price` fields are **read-only**, but `OnHand` can be changed.

VB

```
' The following statement will not compile, because Name is a key
' property and its value cannot be changed.
' prod8.Name = "clamps"

' OnHand is not a Key property. Its value can be changed.
prod8.OnHand = 22
```

Anonymous Types from Query Expressions

Query expressions do not always require the creation of anonymous types. When possible, they use an existing type to hold the column data. This occurs when the query returns either whole records from the data source, or only one field from each record. In the following code examples, `customers` is a collection of objects of a `Customer` class. The class has many properties, and you can include one or more of them in the query result, in any order. In the first two examples, no anonymous types are required because the queries select elements of named types:

- `custs1` contains a collection of strings, because `cust.Name` is a string.

  VB

  ```
  Dim custs1 = From cust In customers
               Select cust.Name
  ```

- `custs2` contains a collection of `Customer` objects, because each element of `customers` is a `Customer` object, and the whole element is selected by the query.

  VB

  ```
  Dim custs2 = From cust In customers
               Select cust
  ```

However, appropriate named types are not always available. You might want to select customer names and addresses for one purpose, customer ID numbers and locations for another, and customer names, addresses, and order histories for a third. Anonymous types enable you to select any combination of properties, in any order, without first declaring a new

named type to hold the result. Instead, the compiler creates an anonymous type for each compilation of properties. The following query selects only the customer's name and ID number from each `Customer` object in `customers`. Therefore, the compiler creates an anonymous type that contains only those two properties.

VB

```
Dim custs3 = From cust In customers
             Select cust.Name, cust.ID
```

Both the names and the data types of the properties in the anonymous type are taken from the arguments to **Select**, `cust.Name` and `cust.ID`. The properties in an anonymous type that is created by a query are always key properties. When `custs3` is executed in the following **For Each** loop, the result is a collection of instances of an anonymous type with two key properties, `Name` and `ID`.

VB

```
For Each selectedCust In custs3
    Console.WriteLine(selectedCust.ID & ": " & selectedCust.Name)
Next
```

The elements in the collection represented by `custs3` are strongly typed, and you can use IntelliSense to navigate through the available properties and to verify their types.

For more information, see Introduction to LINQ in Visual Basic.

Deciding Whether to Use Anonymous Types

Before you create an object as an instance of an anonymous class, consider whether that is the best option. For example, if you want to create a temporary object to contain related data, and you have no need for other fields and methods that a complete class might contain, an anonymous type is a good solution. Anonymous types are also convenient if you want a different selection of properties for each declaration, or if you want to change the order of the properties. However, if your project includes several objects that have the same properties, in a fixed order, you can declare them more easily by using a named type with a class constructor. For example, with an appropriate constructor, it is easier to declare several instances of a `Product` class than it is to declare several instances of an anonymous type.

VB

```
' Declaring instances of a named type.
Dim firstProd1 As New Product("paperclips", 1.29)
Dim secondProd1 As New Product("desklamp", 28.99)
Dim thirdProd1 As New Product("stapler", 5.09)

' Declaring instances of an anonymous type.
Dim firstProd2 = New With {Key .Name = "paperclips", Key .Price = 1.29}
Dim secondProd2 = New With {Key .Name = "desklamp", Key .Price = 28.99}
Dim thirdProd2 = New With {Key .Name = "stapler", Key .Price = 5.09}
```

Another advantage of named types is that the compiler can catch an accidental mistyping of a property name. In the

previous examples, `firstProd2`, `secondProd2`, and `thirdProd2` are intended to be instances of the same anonymous type. However, if you were to accidentally declare `thirdProd2` in one of the following ways, its type would be different from that of `firstProd2` and `secondProd2`.

VB

```
' Dim thirdProd2 = New With {Key .Nmae = "stapler", Key .Price = 5.09}
' Dim thirdProd2 = New With {Key .Name = "stapler", Key .Price = "5.09"}
' Dim thirdProd2 = New With {Key .Name = "stapler", .Price = 5.09}
```

More importantly, there are limitations on the use of anonymous types that do not apply to instances of named types. `firstProd2`, `secondProd2`, and `thirdProd2` are instances of the same anonymous type. However, the name for the shared anonymous type is not available and cannot appear where a type name is expected in your code. For example, an anonymous type cannot be used to define a method signature, to declare another variable or field, or in any type declaration. As a result, anonymous types are not appropriate when you have to share information across methods.

An Anonymous Type Definition

In response to the declaration of an instance of an anonymous type, the compiler creates a new class definition that contains the specified properties.

If the anonymous type contains at least one key property, the definition overrides three members inherited from Object: Equals, GetHashCode, and ToString. The code produced for testing equality and determining the hash code value considers only the key properties. If the anonymous type contains no key properties, only ToString is overridden. Explicitly named properties of an anonymous type cannot conflict with these generated methods. That is, you cannot use `.Equals`, `.GetHashCode`, or `.ToString` to name a property.

Anonymous type definitions that have at least one key property also implement the System.IEquatable(Of T) interface, where `T` is the type of the anonymous type.

For more information about the code created by the compiler and the functionality of the overridden methods, see Anonymous Type Definition (Visual Basic).

See Also

Object Initializers: Named and Anonymous Types (Visual Basic)
Local Type Inference (Visual Basic)
Introduction to LINQ in Visual Basic
How to: Infer Property Names and Types in Anonymous Type Declarations (Visual Basic)
Anonymous Type Definition (Visual Basic)
Key (Visual Basic)

How to: Infer Property Names and Types in Anonymous Type Declarations (Visual Basic)

Visual Studio 2015

Anonymous types provide no mechanism for directly specifying the data types of properties. Types of all properties are inferred. In the following example, the types of `Name` and `Price` are inferred directly from the values that are used to initialize them.

VB

```
' Variable product is an instance of a simple anonymous type.
Dim product = New With {Key .Name = "paperclips", .Price = 1.29}
```

Anonymous types can also infer property names and types from other sources. The sections that follow provide a list of the circumstances where inference is possible, and examples of situations where it is not.

Successful Inference

Anonymous types can infer property names and types from the following sources:

- From variable names. Anonymous type `anonProduct` will have two properties, `productName` and `productPrice`. Their data types will be those of the original variables, **String** and **Double**, respectively.

  VB

  ```
  Dim productName As String = "paperclips"
  Dim productPrice As Double = 1.29
  Dim anonProduct = New With {Key productName, Key productPrice}

  ' To create uppercase variable names for the new properties,
  ' assign variables productName and productPrice to uppercase identifiers.
  Dim anonProduct1 = New With {Key .Name = productName, Key .Price = productPrice}
  ```

- From property or field names of other objects. For example, consider a `car` object of a `CarClass` type that includes `Name` and `ID` properties. To create a new anonymous type instance, `car1`, with `Name` and `ID` properties that are initialized with the values from the `car` object, you can write the following:

  VB

  ```
  Dim car1 = New With {Key car.Name, Key car.ID}
  ```

  The previous declaration is equivalent to the longer line of code that defines anonymous type `car2`.

VB

```
Dim car2 = New With {Key .Name = car.Name, Key .ID = car.ID}
```

- From XML member names.

VB

```
Dim books = <Books>
                <Book Author="Jesper Aaberg">
                    Advanced Programming Methods
                </Book>
            </Books>
Dim anon = New With {books...<Book>}
```

The resulting type for `anon` would have one property, `Book`, of type IEnumerable(Of XElement).

- From a function that has no parameters, such as `SomeFunction` in the following example.

```
Dim sc As New SomeClass

Dim anon1 = New With {Key sc.SomeFunction()}
```

The variable `anon2` in the following code is an anonymous type that has one property, a character named `First`. This code will display a letter "E," the letter that is returned by function First(Of TSource).

VB

```
Dim aString As String = "Example String"
Dim anon2 = New With {Key aString.First()}
' The variable anon2 has one property, First.
Console.WriteLine(anon2.First)
```

Inference Failures

Name inference will fail in many circumstances, including the following:

- The inference derives from the invocation of a method, a constructor, or a parameterized property that requires arguments. The previous declaration of `anon1` fails if `someFunction` has one or more arguments.

```
' Not valid.

' Dim anon3 = New With {Key sc.someFunction(someArg)}
```

Assignment to a new property name solves the problem.

```
' Valid.
```

```
Dim anon4 = New With {Key .FunResult = sc.someFunction(someArg)}
```

- The inference derives from a complex expression.

```
Dim aString As String = "Act "
' Not valid.
' Dim label = New With {Key aString & "IV"}
```

The error can be resolved by assigning the result of the expression to a property name.

VB

```
' Valid.
Dim label1 = New With {Key .someLabel = aString & "IV"}
```

- Inference for multiple properties produces two or more properties that have the same name. Referring back to declarations in earlier examples, you cannot list both `product.Name` and `car1.Name` as properties of the same anonymous type. This is because the inferred identifier for each of these would be `Name`.

```
' Not valid.

' Dim anon5 = New With {Key product.Name, Key car1.Name}
```

The problem can be solved by assigning the values to distinct property names.

VB

```
' Valid.
Dim anon6 = New With {Key .ProductName = product.Name, Key .CarName = car1.Name}
```

Note that changes in case (changes between uppercase and lowercase letters) do not make two names distinct.

```
Dim price = 0

' Not valid, because Price and price are the same name.

' Dim anon7 = New With {Key product.Price, Key price}
```

- The initial type and value of one property depends on another property that is not yet established. For example, `.IDName = .LastName` is not valid in an anonymous type declaration unless `.LastName` is already initialized.

```
' Not valid.

' Dim anon8 = New With {Key .IDName = .LastName, Key .LastName = "Jones"}
```

In this example, you can fix the problem by reversing the order in which the properties are declared.

VB

```
' Valid.
```

```
Dim anon9 = New With {Key .LastName = "Jones", Key .IDName = .LastName}
```

- A property name of the anonymous type is the same as the name of a member of Object. For example, the following declaration fails because `Equals` is a method of Object.

```
' Not valid.

' Dim relationsLabels1 = New With {Key .Equals = "equals", Key .Greater = _

' "greater than", Key .Less = "less than"}
```

You can fix the problem by changing the property name:

VB

```
' Valid
Dim relationsLabels2 = New With {Key .EqualString = "equals",
                                 Key .GreaterString = "greater than",
                                 Key .LessString = "less than"}
```

See Also

Object Initializers: Named and Anonymous Types (Visual Basic)
Local Type Inference (Visual Basic)
Anonymous Types (Visual Basic)
Key (Visual Basic)

Anonymous Type Definition (Visual Basic)

Visual Studio 2015

In response to the declaration of an instance of an anonymous type, the compiler creates a new class definition that contains the specified properties for the type.

Compiler-Generated Code

For the following definition of `product`, the compiler creates a new class definition that contains properties `Name`, `Price`, and `OnHand`.

VB

```
' Variable product is an instance of an anonymous type.
Dim product = New With {Key .Name = "paperclips", Key .Price = 1.29, .OnHand = 24}
```

The class definition contains property definitions similar to the following. Notice that there is no `Set` method for the key properties. The values of key properties are read-only.

VB

```
Public Class $Anonymous1
    Private _name As String
    Private _price As Double
    Private _onHand As Integer
     Public ReadOnly Property Name() As String
        Get
            Return _name
        End Get
    End Property

    Public ReadOnly Property Price() As Double
        Get
            Return _price
        End Get
    End Property

    Public Property OnHand() As Integer
        Get
            Return _onHand
        End Get
        Set(ByVal Value As Integer)
            _onHand = Value
        End Set
    End Property

End Class
```

In addition, anonymous type definitions contain a default constructor. Constructors that require parameters are not permitted.

If an anonymous type declaration contains at least one key property, the type definition overrides three members inherited from Object: Equals, GetHashCode, and ToString. If no key properties are declared, only ToString is overridden. The overrides provide the following functionality:

- **Equals** returns **True** if two anonymous type instances are the same instance, or if they meet the following conditions:
    - They have the same number of properties.
    - The properties are declared in the same order, with the same names and the same inferred types. Name comparisons are not case-sensitive.
    - At least one of the properties is a key property, and the **Key** keyword is applied to the same properties.
    - Comparison of each corresponding pair of key properties returns **True**.

  For example, in the following examples, **Equals** returns **True** only for `employee01` and `employee08`. The comment before each line specifies the reason why the new instance does not match `employee01`.

VB

```
Dim employee01 = New With {Key .Name = "Bob", Key .Category = 3, .InOffice = False}
```

```
' employee02 has no InOffice property.
Dim employee02 = New With {Key .Name = "Bob", Key .Category = 3}

' The first property has a different name.
Dim employee03 = New With {Key .FirstName = "Bob", Key .Category = 3, .InOffice =
False}

' Property Category has a different value.
Dim employee04 = New With {Key .Name = "Bob", Key .Category = 2, .InOffice = False}

' Property Category has a different type.
Dim employee05 = New With {Key .Name = "Bob", Key .Category = 3.2, .InOffice =
False}

' The properties are declared in a different order.
Dim employee06 = New With {Key .Category = 3, Key .Name = "Bob", .InOffice = False}

' Property Category is not a key property.
Dim employee07 = New With {Key .Name = "Bob", .Category = 3, .InOffice = False}

' employee01 and employee 08 meet all conditions for equality. Note
' that the values of the non-key field need not be the same.
Dim employee08 = New With {Key .Name = "Bob", Key .Category = 2 + 1, .InOffice =
True}

' Equals returns True only for employee01 and employee08.
Console.WriteLine(employee01.Equals(employee08))
```

- **GetHashcode** provides an appropriately unique GetHashCode algorithm. The algorithm uses only the key properties to compute the hash code.
- **ToString** returns a string of concatenated property values, as shown in the following example. Both key and non-key properties are included.

VB

```
Console.WriteLine(employee01.ToString())
Console.WriteLine(employee01)
' The preceding statements both display the following:
' { Name = Bob, Category = 3, InOffice = False }
```

Explicitly named properties of an anonymous type cannot conflict with these generated methods. That is, you cannot use `.Equals`, `.GetHashCode`, or `.ToString` to name a property.

Anonymous type definitions that include at least one key property also implement the System.IEquatable(Of T) interface, where `T` is the type of the anonymous type.

Note

Anonymous type declarations create the same anonymous type only if they occur in the same assembly, their properties have the same names and the same inferred types, the properties are declared in the same order, and the same properties are marked as key properties.

See Also

Anonymous Types (Visual Basic)
How to: Infer Property Names and Types in Anonymous Type Declarations (Visual Basic)