What's the big deal about OOP?

Object Oriented Programming (OOP) is certainly a hot topic. Nearly every modern programming language claims to be at least partially OOP-compliant. VB was one of the first to make this claim. It certainly has some OOP features, but falls short of being a truly OOP environment. To understand why, we need to know what 'true' OOP is. It is touted as the next evolution in program design. It is supposed to make programs behave more like the 'real world.' It promises better productivity and code re-use.

A review of some OOP terms

We have already used some of these terms freely as VB programmers, but we ought to pay close attention to what they mean in OOP vernacular.

Object:

Our previous working definition as something that contains both data and code is still acceptable. Remember that while controls are objects, all objects are not controls. We will be defining some objects that do NOT have a visible component. Not all objects will have properties, events, methods, or visible interfaces. We'll keep defining this term all semester.

class:

This term defines a TYPE of object. It is analogous to the TYPE ... END TYPE structure in straight VB. It defines a type of object, but not an instance of the object. If we were defining a kind of car, the factory that generates the car model might relate to the class. The actual cars are NOT the class.

instance:

An instance is generated by a class. If a class was like a TYPE structure, an instance is like the variable DIMmed to be the type. In our factory analogy, the actual cars would be instances of the car type.

properties:

The characteristics of an object that are exposed to users of the object. NOT all variables in an object definition are properties. Some OOP languages do not define properties at all. They are not really necessary. In VB, properties look like variables, but they are in fact 'hidden' and 'protected' variables that can only be accessed through special methods.

methods:

These are the code segments that describe things an object can do. Most of the actual code writing in OOP has to do with writing methods for your various objects.

events:

Events are messages the object can raise when a particular condition is met. The object's programmer does NOT decide what happens in event code, but exposes the events to the object's user (who is usually another programmer). Events are a relatively advanced idea, and we will
Example: a simple car class

**Properties of the car class:**
- speed
- direction
- engineType
- frameType

**Methods of the car class:**
- accelerate()
- decelerate()
- turnLeft()
- turnRight()

**How the car might be used**

```vbnet
DIM myCar as Car
SET myCar = new Car
myCar.accelerate()
msgBox myCar.speed
```

Note that after we create a variable of type car, we can access its properties and methods through the drop-down completion box just as if it had been a built-in class. Note also that we do not deal with the class (Car) directly, but with an instance of the class (myCar). This is familiar already. We don't deal with the textbox class, but with individual instances of it. (txtInput.text)

**The OOP words to know**

**Encapsulation**

*Code and data should be made as local as possible.*

You shouldn't need to know how every part of the car's engine works.
If you learn an interface (the steering wheel, accelerator, and displays) you can manage the car without knowing exactly how the fuel injector works.
Test the code thoroughly, then don't worry about it.
Design procedures for flexibility, re-use.
VB encourages encapsulation through modules.
Our strategy of making all variables as local as possible is also a facet of encapsulation.
VB (as we have done it so far) encourages encapsulation, but does not require it.
True OOP lets you think of objects, with all the characteristics and behaviors of the object 'hidden' also known as 'data hiding'

**Polymorphism**

This is a fancy word for a simple idea:

*The same thing can be done in different ways.*

In the physical world, the term 'start' might apply to a car, lawnmower, or motorcycle. In OOP terms, 'start' would be a method of all three objects. (remember that methods are actions the object knows how to do). The start method would generally do the same thing in each object, but the underlying behavior would be different. For example, to start a car, you turn a key. To start a motorcycle, you might kick a pedal. To start a lawnmower, you would pull a lanyard. When you look closely at the object, the exact way the object is started is important. However, when you are writing an algorithm that uses the object, you don't really care. We might invoke lawnMower.start(), motorCycle.start(), or car.start(). It would be up to the individual objects to determine how the start method will actually happen.

Polymorphism is often also used to allow one function to work on multiple parameter types or lists. For example, the msgBox function you are used to takes five parameters. Almost nobody uses all five. We frequently will call this procedure with only one parameter. The ability to still function with different types of parameters is a frequently seen kind of polymorphism.

VB uses the variant data type to simulate polymorphism, and it supports some other kinds of polymorphism as well.

**Inheritance**

Inheritance is one of the most important ideas in OOP. Unfortunately, it is the one VB has the most trouble with. The basic idea is quite simple.

Imagine the car class we discussed above. This would be quite useful on its own, as we could generate new instances of the class to make several different cars. However, what if we wanted to create a police car? It ought to start with the same characteristics as a regular car, but then should add some special features that relate to cop cars, such as a floodLightOn property and a soundSiren() method.

We could copy and paste all the features of the original car class to a new blank class, then add the new features, but this seems like a waste. It would be much more like the real world if we could take the car class 'straight from the factory' and work on only the modifications needed to make it a police car class. This is what inheritance promises. You can build new classes from existing classes, so you don't have to start from scratch every time.

Windows shows some evidence of inheritance in its design. If you look closely at the VB controls, they tend to have very similar property and event lists. In particular, the picture box and the form object have strikingly similar lists of properties, events, and methods. This is because they both inherit the same characteristics from a common parent, the windows Rectangle class (which is hidden from us by VB). Have you ever noticed that the command button has a backColor property, but that changing the property has no effect? Why was this property included? Because it was INHERITED from a parent class. The default behavior (of allowing the background color to be changed) was over-ridden by the Microsoft standard that command buttons are always the same color, defined in the user settings.
VB shows evidence of Window's inheritance features, although it allows very limited support for inheritance in the objects you create within the language.

The Air Traffic Control Sim

As a simple example of OOP design as it relates to VB, we will write a (very much simplified) simulation of an Air Traffic Control (ATC) station. What we want to do is design an environment that allows the controller (the program's user) to send simple commands to an aircraft (or to many aircraft) and control the landing, take-offs, and flights through an airspace. Eventually we will want to see this visually, but we will first look at various ways of encapsulating an aircraft object.

So why OOP?

That's a really good question. It is quite possible to write this simulation without using any formal classes at all (in fact, I did so in previous versions of VB that did not support classes). I am hopeful that as we progress through this example you will see why the OOP approach makes a certain amount of sense for this project.

What objects will we create?

The first part of OOP design is trying to determine exactly what objects you will create, and what properties, methods, and events the objects will need. In this case, it is relatively obvious that I need an aircraft object, as my simulation is primarily about analyzing the behaviour of airplanes. I might also want to make objects for the airport(s), the controller, and the airspace. For this demonstration, we will only 'objectify' the aircraft. You will see why it might make sense to make objects of the other players as well, but let's keep our code focused for now on simply making an aircraft object.

What characteristics should the plane have?

The way you determine which properties and methods an object will have is by imagining what the user will want. Remember, we will be encapsulating the code so the only way the outside program can manipulate the object is by playing with the properties and methods. Eventually, I think the controller will need to know about the plane's altitude, speed, and direction, so these sound like promising properties. We want to be able to tell the plane to land, take off, turn left, turn right, and change altitude up and down. These would relate to methods. Let's not worry about the events this week.

Properties the 'old fashioned way'

(Object Oriented Programming the way Grandma used to do it...)

Examine the following module:

```vbnet
Type planeType
    altitude As Integer
    airspeed As Integer
    direction As Integer
End Type
```
It is attached to this form:

...and here is the associated code:

Dim myPlane As planeType

Private Sub scrAlt_Change()
    myPlane.altitude = scrAlt.Value
End Sub

Private Sub scrDirec_Change()
    myPlane.direction = scrDirec.Value
End Sub

Private Sub scrSpeed_Change()
    myPlane.airspeed = scrSpeed.Value
End Sub

Private Sub Timer1_Timer()
    Dim temp As String
    temp ="
    temp = temp + "airspeed:" + Str(myPlane.airspeed) + vbCrLf
    temp = temp + "altitude:" + Str(myPlane.altitude) + vbCrLf
    temp = temp + "direction:" + Str(myPlane.direction) + vbCrLf

    lblOutput.Caption = temp
End Sub

Notes on the example:

- This is not OOP. We are approximating with the best technology we had in the beginning course or older versions of VB. (which is the type structure).
- The characteristics such as airspeed and altitude 'feel' like properties.
- Once we have defined the plane type, we can automatically get to its fields in the editor.
- There are some drawbacks, though...
  - The programmer could assign illegal values (airspeed could be -100!)
This is not how planes work... You can tell them to accelerate or decelerate, but you can’t just arbitrarily set the speed.

We want the program to be able to tell the plane 'turn left' and have the plane respond appropriately.

What's the deal with the directions being integers?

**Enum**

Let's answer that last question right here. To simplify our life, let's just say there will be eight directions the plane can head, NW, N, NE, E, SE, S, and SW. Since we will be referring to those values a lot, it might be nice if we could keep track of them easily. Take a look at the following changes to the module where we defined planeType:

```vbnet
Public Enum DirecTypes
    NORTHWEST = 0
    NORTH = 1
    NORTHEAST = 2
    EAST = 3
    SOUTHEAST = 4
    SOUTH = 5
    SOUTHWEST = 6
    WEST = 7
End Enum

Type planeType
    altitude As Integer
    airspeed As Integer
    direction As DirecTypes
End Type
```

Now look at what happens when we attempt to assign a value to myPlane.direction from the editor:

```
myPlane.direction = 
nd Sub
Private Sub scrAlt
myPlane.altitude
d Sub
Private Sub scrDire
myPlane.direction
```

as you can see, a list of legal DirecTypes values appears. An enumeration is simply a special list of constants. It is used in situations like this to protect the user from dealing with the actual values and giving them an easier name to work with. Note that the values are still integers, and we can use them as such (in fact we are when we are assigning values from the scroll bars) but we also can think in terms of these pre-defined constants. Enums are very handy in this kind of situation, as you will see.
Let's try to have some class, here.

The problems we discussed above (little control over what went into the fields, and no way to control the objects behavior through methods) were the historical impetus for the development of object oriented programming. Let's take our module, and re-create it as a new kind of module called a class module.

In the project menu, one of the choices is 'class module'. Let's add that sonofagun. Don't use the class file interface wizard yet!! We'll try it by hand, because we don't always trust the wizards. - Here's my first effort:

```vba
Public altitude As Integer
Public direction As DirecTypes
Public x As Integer
Public y As Integer
Public moving As Boolean
Private speedVal As Integer 'how many units to move each time

Public Enum DirecTypes
  NORTHWEST = 0
  NORTH = 1
  NORTHEAST = 2
  EAST = 3
  SOUTHEAST = 4
  SOUTH = 5
  SOUTHWEST = 6
  WEST = 7
End Enum
```

Notice what I have done. I started with the Enum exactly as it was, and I added a set of public module-level variables. We have been ignoring the public/private declarations up to now, but they are important in this case. Note that I changed my variables a bit, but most of them are public. This means that the programmer who uses this object can get to the variable as a property. I'll save this as 'lazyPlane.cls' (because I'm making a plane, but I'm doing properties the lazy way). Now in my form, I can generate a variable of type lazyPlane, and set it to a new lazyPlane. Then when I try to access the variable, I will have access to its properties.

**So why is this better than a type-definition?**

So far, it isn't. It still has the same major weaknesses, that: Properties can be changed without any error trapping, and there are no built-in methods.
To improve the latter situation, we can add a bunch of public subs, like this:

```vba
Public Sub turnRight()
  Me.direction = Me.direction + 1
  If Me.direction > 7 Then
    Me.direction = 0
  End If
End Sub
```
Public Sub turnLeft()
    Me.direction = Me.direction - 1
    If Me.direction < 0 Then
        Me.direction = 7
    End If
End Sub

Public Sub land()
    If Me.altitude > 5 Then
        MsgBox "you are too high to land"
    Else
        Me.moving = False
        Me.altitude = 0
    End If
End Sub

Public Sub takeOff()
    Me.moving = True
    Me.altitude = 5
End Sub

Public Sub move()
    'analyzes direction, generates new values for x and y
    If Me.moving = True Then
        Select Case Me.direction
            Case NORTH
                Me.x = Me.x + 0
                Me.y = Me.y + speedVal
            Case NORTHEAST
                Me.x = Me.x + speedVal
                Me.y = Me.y + speedVal
            Case EAST
                Me.x = Me.x + speedVal
                Me.y = Me.y + 0
            Case SOUTHEAST
                Me.x = Me.x + speedVal
                Me.y = Me.y - speedVal
            Case SOUTH
                Me.x = Me.x + 0
                Me.y = Me.y - speedVal
            Case SOUTHWEST
                Me.x = Me.x - speedVal
                Me.y = Me.y - speedVal
            Case WEST
                Me.x = Me.x - speedVal
                Me.y = Me.y + 0
            Case NORTHWEST
                Me.x = Me.x - speedVal
                Me.y = Me.y + speedVal
        End Select
End If
End Sub

If we try to use this object in a form, you will see that the properties and methods do show up.

Private Sub Form_Load()
   Dim plane As lazyPlane
   Set plane = New lazyPlane
   plane.
End Sub

Now, we are beginning to have what can be considered an object.

Let's take a closer look at what is happening in the code above:

**turnRight(), turnLeft()**

These methods are very similar. They cause the plane to change direction by one value. Note how they act as an interface to the direction property. The code is responsible for turning the airplane correctly. By attaching these methods, we are approaching the definition of an object we had been seeking: a combination of data (the properties) and code (the methods). The user of this object now does not have to set the direction manually. Instead, she can tell the plane object to 'turn left' and it will know how to maneuver the properties properly. In fact, now that we have the turnLeft and turnRight procedures, we might not want to allow the user to directly set the direction property, as it might cause problems. As long as the initial value of direction is legal, and the user is only allowed to manipulate the direction value through turnLeft and turnRight methods, we will never get an illegal value in direction. This is a primary benefit of encapsulation.

**land(), takeOff()**

These methods are a form of data abstraction. The air traffic controller will not usually be looking at properties directly, but will want to issue command such as take off and land to the aircraft. We want to establish an interface so these commands can be issued, tested, and can manipulate the appropriate properties. Take off and land are appropriately thought of as methods, not properties, as they involve some action the object can do. Their main job, however, is the maintainance of property values. The user (which might be a program) can invoke natural methods, and not worry much about how these methods might interact with properties.

**move()**

The move method also embodies typical OOP philosophy. It tries to anticipate some command the user (the controller or a program) might invoke, and appropriately modify the properties. I introduced here X and Y values. I did make an assumption that the plane would be flying in some kind of standard cartesian coordinate system (which is NOT the default in VB).
Note that I hid all the implementation details of the move command. The controller should just tell the airplane to move, and it should be able to move in the appropriate direction, and modify its properties accordingly.

Public and private properties

(Sounds like some kind of law seminar...)

Our aircraft class now has methods as well as properties, so it is useable, but it still has some significant problems. Those of you who already know something about OOP were probably choking at my use of public variables as properties. In programming circles, this practice is considered about as polite as dropping a live muskrat into a teapot. It can be done, but it's very rude and will likely lead to a big mess.

Remember, we considered variable scope to be a part of encapsulation. We are interested in keeping our variables as local as possible. In a class declaration, a public variable is WIDE open, and can be changed by anybody, with very unpredictable results. We do not want this behavior. Somehow, we need to protect our properties so they cannot be changed without some degree of monitoring. Also, some properties (like direction, perhaps) ought to be read-only. That is, a program that uses the class can read the property value, but cannot set it on the fly.

One solution is to have all of the class level variables (often called *instance variables*) listed as private. This is much safer than having them public, but now they are not accessible as properties.

In Java, it is common to have all instance variables set as private. The class designer then uses methods to allow access to the instance variables. In essence, such a class would have no properties at all, just methods. The methods could be used to set the properties, but the properties could not be set directly. A set of these methods for changing 'property' values is called 'setters' and another set designed to retrieve these values is called 'getters'.

In VB, we can define special methods very much like these setters and getters, and use them to generate properties.

Here's the key idea:

**Properties are really methods acting on invisible private instance variables!!!**

(No need to shout!)

To the user of the class, the properties will feel just like the public variables we have seen, but there are some important distinctions.

Here's another version of the Aircraft class general area (or part of it, anyway)

'local variable(s) to hold property value(s)
Private mvaraltitude As Integer 'local copy
Private mvardirection As DirecTypes 'local copy
Private mvarx As Integer 'local copy
Private mvary As Integer 'local copy
Private mvarmoving As Boolean ' local copy
Private speedVal As Integer ' how many units to move each time

Notice that all my instance variables are now **private** and their names have been changed to mvarsomething. We'll talk about that in a minute.

Here's a new set of methods I have added:

Public Property Get direction() As DirecTypes
' used when retrieving value of a property, on the right side of an assignment.
' Syntax: Debug.Print X.direction
   direction = mvardirection
End Property

Public Property Let direction(ByVal vData As DirecTypes)
' used when assigning a value to the property, on the left side of an assignment.
' Syntax: X.direction = 5
   if (vData < 0) or (vData > 7) then
      msgBox "Illegal value! Setting direction to NORTH"
      mvardirection = NORTH
   else
      mvardirection = vData
   end if
End Property

**So what did this do for us?**

Well, notice that direction NO LONGER EXISTS!!! It is now a 'virtual variable'. In its place, I have a private instance variable called **mvardirection** which cannot be changed directly from the outside.

The **property get** procedure returns back the value of mvardirection, but does not allow the variable to be changed directly. If I wanted to have a read-only property, I would give it a **property get** procedure without a **property let**.

The **property Let** procedure is far more interesting, and illustrates why properties are sometimes better than public instance variables. When direction was public, the calling program could set direction to ANY integer value, including illegal values such as -1 and 8. There was no way to prevent this.

I added error-checking code to the Let procedure, which accepts a value in a temporary variable called vData. I then checked vData with an if statement to ensure the value was legal. If not, I warned the user with a messagebox (great during debugging, but probably not such a swift idea in a final release) and set the value to something legal. It is now impossible to have an illegal value in direction.

Note that my original code for turnLeft and turnRight will no longer work, because they incremented the property to illegal values **before** testing them. I now have to change them slightly, but I'm glad, because this means I can trust this property to always have a legal value. Here's the new code for turnLeft():

Public Sub turnLeft()
   If Me.direction > 0 Then
      Me.direction = Me.direction - 1
   Else
      Me.direction = 7
End Sub
We're off to see the Wizard...

As you can tell, the exact syntax of the *property get* and *property let* procedures can get a little bit hairy. Fortunately, VB does provide an alternative. When you choose to add a class module to your project, one of the choices is VB class builder. When you select this, a wizard will guide you through the creation of your class file. This is what the class builder looks like after I have finished adding in all the characteristics for Aircraft:

![Class Builder Image](image)

As you look at the buttons, you will see there are tools for creating new classes and collections, and for adding properties, methods, events, and enums to an existing object. I won't insult you by going step-by-step, but you can get the main idea by looking at the dialog for adding the altitude property.
This view shows AFTER the property has been created, so it cannot be modified, but it shows how you could add properties and change their characteristics. The attributes screen lets you type in a simple help message that will pop up in the object browser.

This is very convenient, but you may still need to edit the code by hand.

Finally, here is the complete code for the Aircraft class, as generated by the class builder and modified by me:

'a aircraft class
'a simple class file designed to model
'an aircraft in an Air Traffic Control simulation
'Andy Harris, 2/99

'local variable(s) to hold property value(s)
Private mvaraltitude As Integer 'local copy
Private mvardirection As DirecTypes 'local copy
Private mvarx As Integer 'local copy
Private mvary As Integer 'local copy
Private mvarmoving As Boolean 'local copy
Private speedVal As Integer 'how many units to move each time

Public Enum DirecTypes
    NORTHWEST = 0
    NORTH = 1
    NORTHEAST = 2
    EAST = 3
    SOUTHEAST = 4
    SOUTH = 5
SOUTHWEST = 6
WEST = 7
End Enum

Public Sub turnRight()
    If Me.direction < 7 Then
        Me.direction = Me.direction + 1
    Else
        Me.direction = 0
    End If
End Sub

Public Sub turnLeft()
    If Me.direction > 0 Then
        Me.direction = Me.direction - 1
    Else
        Me.direction = 7
    End If
End Sub

Public Sub land()
    If Me.altitude > 5 Then
        MsgBox "you are too high to land"
    Else
        Me.moving = False
        Me.altitude = 0
    End If
End Sub

Public Sub takeOff()
    Me.moving = True
    Me.altitude = 5
End Sub

Public Sub move()
    'analyzes direction, generates new values for x and y
    'expects CARTESIAN coordinates (Y is larger at top of page)
    If (Me.moving = True) Then
        Select Case Me.direction
        Case NORTH
            Me.x = Me.x + 0
            Me.y = Me.y + speedVal
        Case NORTHEAST
            Me.x = Me.x + speedVal
            Me.y = Me.y + speedVal
        Case EAST
            Me.x = Me.x + speedVal
            Me.y = Me.y + 0
        Case SOUTHEAST
            Me.x = Me.x + speedVal
        End Select
Me.y = Me.y - speedVal
Case SOUTH
  Me.x = Me.x + 0
  Me.y = Me.y - speedVal
Case SOUTHWEST
  Me.x = Me.x - speedVal
  Me.y = Me.y - speedVal
Case WEST
  Me.x = Me.x - speedVal
  Me.y = Me.y + 0
Case NORTHWEST
  Me.x = Me.x - speedVal
  Me.y = Me.y + speedVal
End Select
End If
End Sub

Public Property Let moving(ByVal vData As Boolean)
  'used when assigning a value to the property, on the left side of an assignment.
  'Syntax: X.moving = 5
  mvarmoving = vData
End Property

Public Property Get moving() As Boolean
  'used when retrieving value of a property, on the right side of an assignment.
  'Syntax: Debug.Print X.moving
  moving = mvarmoving
End Property

Public Property Let y(ByVal vData As Integer)
  'used when assigning a value to the property, on the left side of an assignment.
  'Syntax: X.y = 5
  mvary = vData
End Property

Public Property Get y() As Integer
  'used when retrieving value of a property, on the right side of an assignment.
  'Syntax: Debug.Print X.y
  y = mvary
End Property

Public Property Let x(ByVal vData As Integer)
  'used when assigning a value to the property, on the left side of an assignment.
  'Syntax: X.x = 5
  mvarx = vData
End Property

Public Property Get x() As Integer
  'used when retrieving value of a property, on the right side of an assignment.
  'Syntax: Debug.Print X.x
  x = mvarx
End Property
Public Property Let direction(ByVal vData As DirecTypes)
'used when assigning a value to the property, on the left side of an assignment.
'Syntax: X.direction = 5
    If (vData < 0) Or (vData > 7) Then
        MsgBox "vData is " + Str(vData)
        MsgBox "Illegal value! Setting direction to NORTH"
        mvardirection = NORTH
    Else
        mvardirection = vData
    End If
End Property

Public Property Get direction() As DirecTypes
'used when retrieving value of a property, on the right side of an assignment.
'Syntax: Debug.Print X.direction
    direction = mvardirection
End Property

Public Property Let altitude(ByVal vData As Integer)
'used when assigning a value to the property, on the left side of an assignment.
'Syntax: X.altitude = 5
    mvaraltitude = vData
End Property

Public Property Get altitude() As Integer
'used when retrieving value of a property, on the right side of an assignment.
'Syntax: Debug.Print X.altitude
    altitude = mvaraltitude
End Property

Private Sub Class_Initialize()
    speedVal = 5
    Me.direction = NORTH
End Sub

Discussion of the code

There is very little new here, but we ought to describe some things with a little more care.

Class_Initialize()

This method is similar to the form_load in a traditional VB form. It contains code that will occur when the class is generated with the NEW keyword from the calling program. It is similar to a constructor in other OOP languages. VB also includes a Class_Terminate() method that occurs as the class is closing down, similar to destructors in other OOP languages.

property get

Property get procedures are all pretty similar. They usually follow this pattern

Public Property Get propName() As type
where \textit{propName} is the name of the property and \textit{type} is the variable type of the property. A get procedure acts much like a function. It returns back the value of the property. When the property is on the right-hand side of an assignment statement, the get procedure is being accessed.

You rarely have to modify the code as it comes out of the class builder. It will simply return the value of the private instance variable.

Almost every property will have a get procedure. If it does not need to be retrieved by the user, it should probably be a private instance variable rather than a property.

\textbf{property let}

Property let procedures are usually a bit more complex than the get procedures. They usually look something like this:

\begin{verbatim}
Public Property Let propName(ByVal vData As type)
\end{verbatim}

where again \textit{propName} is the name of the property and \textit{type} is a variable type.

Let procedures act more like subprograms with a parameter. They accept a value and temporarily store it in vData. The \textit{ByVal} keyword indicates that if a variable is passed to the procedure, we will work on the \textit{value} of that variable, over-riding VB’s natural inclination to work on a reference to that variable instead.

The code essentially copies the value in vData (a temporary variable) over to the private instance variable. You will almost always want some error checking in your let procedures, to ensure that only appropriate values get sent to your instance variables, and that your property cannot be given a value it cannot deal with.

One common strategy is to make vData a variant type, and then contain code to convert vData into the appropriate type before assigning it to the private variable. It is also common to do some bounds checking to ensure that the values fall in some pre-defined range.

If you want to have a read-only property, remove the property let procedure associated with that property.

\section*{The ATC form}

Here is a form that uses the Aircraft class...

\begin{verbatim}
' frmATC
'An air traffic control simulation
'designed to demonstrate OOP in VB
'andy Harris, 2/99

'create an aircraft instance
Dim plane As New Aircraft

Private Sub cmdHigher_Click()
    plane.altitude = plane.altitude + 5
End Sub
\end{verbatim}
Private Sub cmdLand_Click()
    Dim direcOK As Boolean

    'check the direction
    'this airport has Runways going N, S, E, and W,
    'but NW, NE, SW, SE not legal directions

    direcOK = False
    If plane.direction = NORTH Then
        direcOK = True
    ElseIf plane.direction = SOUTH Then
        direcOK = True
    ElseIf plane.direction = EAST Then
        direcOK = True
    ElseIf plane.direction = WEST Then
        direcOK = True
    End If

    'ensure plane is on airport
    If imgPlane.Left > imgAirport.Left Then
        If imgPlane.Left < (imgAirport.Left + imgAirport.Width - imgPlane.Width) Then
            If imgPlane.Top < imgAirport.Top Then
                If imgPlane.Top > (imgAirport.Top - imgAirport.Height + imgPlane.Height) Then
                    plane.land
                    MsgBox "Nice Landing"
                Else
                    MsgBox "Too far South!"
                End If
            Else
                MsgBox "too far North!"
            End If
        Else
            MsgBox "too far East!"
        End If
    Else
        MsgBox "too far West!"
    End If
End Sub

Private Sub cmdLeft_Click()
    plane.turnLeft
End Sub

Private Sub cmdLower_Click()
    'go lower, but don’t allow them to crash
    If plane.altitude <= 5 Then
        MsgBox "You cannot go lower than 500 feet!!"
    Else
        plane.altitude = plane.altitude - 5
    End If
Private Sub cmdRight_Click()
    plane.turnRight
End Sub

Private Sub cmdTakeOff_Click()
    plane.takeOff
End Sub

Private Sub Form_Load()
    'initialize the form to a normal cartesian system
    frmATC.Scale (-100, 100)-(100, -100)

    'tell the OBJECT where the image box is
    plane.x = imgPlane.Left
    plane.y = imgPlane.Top
End Sub

Private Sub Timer1_Timer()
    Dim temp As String
    'move the OBJECT
    plane.move

    'set up the image box so it has the right picture
    imgPlane.Picture = imgStore(plane.direction).Picture

    'also refer to the object for the location
    imgPlane.Left = plane.x
    imgPlane.Top = plane.y

    'output some stuff
    temp = "X: " + Str(plane.x)
    temp = temp + "  Y: " + Str(plane.y)
    temp = temp + "  Alt: " + Str(plane.altitude) + "00 ft"
    lblOutput.Caption = temp
End Sub

Finally, just to show how flexible this approach is, note how easy it is to use an array of aircraft. I'll only show the timer event, but this will give you a feel for how the code changes.

Private Sub Timer1_Timer()
    Dim temp As String
    Dim i As Integer
    For i = 0 To 1
        plane(i).move
        picPlane(i).Picture = imgStore(plane(i).direction).Picture
        picPlane(i).Left = plane(i).x
        picPlane(i).Top = plane(i).y
        picPlane(i).Cls
        picPlane(i).Print i
    Next i
Next i

temp = Str(currentPlane) + ": "
temp = temp + "X: " + Str(plane(currentPlane).x)
temp = temp + " Y: " + Str(plane(currentPlane).y)
temp = temp + " Alt: " + Str(plane(currentPlane).altitude) + "00 ft"
lblOutput.Caption = temp
End Sub

The full code is available here: