Notes.net

Iris Today

Using the *object-oriented* features of LotusScript

by
Bruce
Perry

Level: Advanced
Works with: All
Updated: 10/01/2001

Object-oriented (OO) development doesn't have to be mysterious. It's really just another way to organize your code. The good news is that if you're writing Notes/Domino applications in LotusScript, you already know how to use objects because the front-end and back-end classes like NotesUIDocument and NotesDatabase are fundamental to everything you do. If, however, you're like most Notes/Domino developers I know, you write your code to use those objects in the built-in subroutines (like QueryOpen and QuerySave) that Domino Designer creates for your design elements, and you also write your own subroutines and functions. That seems to be typical, even for most advanced LotusScript developers, but there's a lot more that you can do with OO to improve your applications.

Have you ever seen a Notes database with five or six versions of the same function in different locations? Or worse, have you ever fixed a problem in one LotusScript function only to discover that you need to fix it elsewhere too? If so, you already know one advantage to defining your own classes. Classes help you keep related code and data together—a strategy that is sometimes called "encapsulation"—so that many changes can be made to your implementation without forcing you to dig through your code to find all the different places that need to be updated.

It's not all that surprising that the object-oriented features of LotusScript aren't being used as often as they could be. Most books and magazine articles tend to touch only lightly on the topic. Then too, some may worry about needing to learn lots of OO theory. However, you can derive immediate and practical benefits from classes and objects without going to all that trouble. Other developers may shy away from it because they have heard (as I did, early on) that LotusScript isn't OO in the "right" way. LotusScript may not have all the OO features of C++ or Java, but the features that are there can be very useful if you know how to take advantage of them.

In this article, we'll develop five useful classes that demonstrate OO techniques. Each new class will build on the previous classes to add additional capabilities. We'll learn how to extend existing classes and data types by composition and by inheritance. Since it's not possible to inherit from the built-in Notes classes in LotusScript, we'll learn how to use composition to get around that limitation and build new classes based on the built-in classes. We'll also learn about base classes and subclasses and how a subclass can override the behavior of the classes it inherits from. We'll see how the use of classes can simplify a Notes database by helping to eliminate duplicate code. Finally, we'll learn how to override events with our classes and thus learn an additional way to share code.

Although covering every nuance of LotusScript's object-oriented features is beyond the scope of one article, we'll look at the most important ones and recommend resources for those who want to know more. I'm going to avoid saying much about the features of other OO languages. Though that knowledge might give you a broader theoretical understanding of object-oriented languages, it won't help you write object-oriented LotusScript tomorrow. For those interested in general information about object-oriented programming, I've listed several books in the **Additional**

resources section.

This article is intended for Notes developers who use LotusScript but who haven't been exposed to object-oriented programming yet. The code for this article was developed and tested in Notes R5. It has received some testing in Notes 4.6 as well. A sample database in the **Iris Sandbox** includes the code described in this article. To get the most out of this article, you'll find it helpful to know about script libraries and events.

Basic definitions

Before we get started, let's look at a few definitions. An object is something with a name, a set of attributes, and a set of methods. Attributes are data values, and methods are pieces of code that can manipulate the data values. A class is a group of dims, subs, and functions that can represent and manipulate the attributes of an object. The dims set up storage for the object's attributes, and the subs and functions contain the code for the object's methods. Classes can also contain a special type of routine, called a property, that is used to get or set a data element that was "dim'ed" using the Private keyword.

In LotusScript, an object is created using the New keyword. You can have many objects of the same class at the same time in your code, and they do not share their data values. The word *instance* is often used instead of (or in addition to) *object* in order to emphasize this fact.

Data values, properties and methods, and methods of an object can only be referenced through "dot notation" using the instance on the left of the dot, and the data value, property, or method on the right. For example, if you have a class named Book with a property called Title, you could have code like this:

```
Dim ThingOne as New Book
Dim ThingTwo as New Book
ThingOne.Title = "The Cat In The Hat"
ThingTwo.Title = "Green Eggs And Ham"
```

By the way, a wise programmer once observed that object-oriented programming sounds a lot less solemn and complicated if you substitute the word *thingy* for the word *object* whenever you see it. Now, let's take a look at some thingy-oriented code.

Creating a simple LotusScript class

First we'll take a look at a simple LotusScript class. On its own, it's not very useful. It's simply a container for a string and a key value to identify it. Please bear with me. In conjunction with some other classes we're about to build, I think you'll see that it's a very handy building block.

Note that the class's data is declared Private, and access is provided via properties rather than direct access to the data elements. Why take this extra step? We do it because the code will inevitably change. With these properties in place, we can validate values users give us via the set properties and calculate values returned by get properties that don't have corresponding data elements. For example, the class could have an additional property that returns a metric dimension even though the internal data element is stored in the English system. Also, by leaving out the set property for a data element, you can protect it from being altered by any user of that class as long as the data element in question is private. We don't get those advantages if we allow direct access to a class's data. In the OO world, providing data access like this via properties is called data hiding. Although it's possible to declare class data elements as public; in general, a class's data should be private. The users of the class should be provided with properties, functions, and subroutines to access only the information they need. This will prevent them from changing the class's data in inappropriate ways.

```
Public Class ListItem
' Note: this class issues an E_BLANK_KEY error if sub new or set key
' is passed a blank key
    Private m_key As String
    Private m_value As String

    Sub new (key As String, value As String)
        If key = "" Then                    'don't allow a blank key
            Error E_BLANK_KEY, E_BLANK_KEY_MSG
        End If

        m_key = key
        m_value = value
    End Sub

    Property Get key As String
        key= m_key
    End Property

    Property Set key As String
        If key = "" Then                    'don't allow a blank key
            Error E_BLANK_KEY, E_BLANK_KEY_MSG
        End If
        m_key = key
    End Property

    Property Get value As String
        value= m_value
    End Property
    Property Set value As String
        m_value = value
    End Property

    Sub PrintItem
            Print "ListItem: key = " & Me.m_key & " value = " &
            Me.m_value
    End Sub

End Class
```

If you look at the Sub New method and the Set Key property, you'll see they both issue an error when they encounter a blank key. Why is that? Simply because that's the best way available to indicate that a problem has occurred. Neither the property nor the method has a return value available for signaling error conditions.

Creating a class using composition

What can we do with the ListItem class? Not much without more classes. Let's create another class so that we can do something useful. LotusScript's list data type is an extremely useful tool. If you've made much use of it though, you may have noticed that there's no way to get a count of the elements in a list without keeping track yourself (or cycling though the list). Here's a class that can be used as a "wrapper" around a list, which solves this problem. (The **BetterList class** sidebar contains an uninterrupted version of the code.)

```
Public Class BetterList
    Private m_list List As Variant
    Private m_count As Integer
```

```
Property Get Count As Integer
    Count = m_count
End Property

Public Function DeleteList
    Erase m_list
End Function

Public Sub new
    m_count = 0
End Sub

Sub Delete
    Call Me.DeleteList
End Sub
```

Notice that the class’s variables all start with the string *m_*. This is a convention I’ve borrowed from the C++ world. It allows us to know at a glance which variables belong to the class itself rather than the current function. I’ve found that this convention can save a lot of time, especially in large classes. Notice also that m_list is private; there’s no way for someone using this class to tinker with the list directly. That’s the only way to be absolutely sure that the count stays correct.

Next are several of BetterList's class functions (or methods). They're just like regular functions, but they're defined within the class.

```
Public Function DeleteItem(key As String) As Integer
    Dim rval As Integer

    'if the key is in the list, erase the object
    If ( Iselement(m_list(key)) ) Then
        Erase m_list(key)
        m_count = m_count-1
        rval = True
    Else
        'if there's no such key, warn of an error
        rval = False
        Print "Item " & key & " not found. It could not be deleted."
    End If

    DeleteItem = rval
End Function

Public Function AddItem(key As String, item As ListItem) As Integer

    'just add the item if the key doesn't exist
    If ( Not Iselement(m_List(key)) ) Then
        Set m_list(key) = item
        m_count = m_count+1
    Else
        'if the key does exist, erase the current object in the list
        'and add the new one
        Erase m_List(key)
        Set m_list(key) = item
    End If

End Function
```

In the next section of code, you’ll see that GetItem returns a variant. Why is that? Well, the function could return a ListItem, but then we’d have to have additional classes just like BetterList that return each type of object that we want to deal with. This way one class can handle multiple object types. Using variants in this way can cause problems if done carelessly. If GetItem returns an item of the wrong type, you'll most likely get a runtime error when you try to treat it as the type you're expecting.

(In the next class we discuss, EnhancedUIDoc, we see one way around this problem. There, we immediately assign the variant to an object of the correct type and any further access is done via the object. This assignment in EnhancedUIDoc also has the effect of revealing errors in property names and functions at compile time rather than runtime. I prefer to use the compiler to find errors whenever I can. It's far easier than finding problems through testing and saves time as well.)

This is the real workhorse function in this class. It's used to get access to the objects we're storing in the list.

Also, just below, you can see that we've put error handling code in place. It indicates the type of error and the class and method (or property) where the error has occurred; it then permits processing to continue. Though basic, this can be quite helpful in pinpointing the origin of an error. Without it, we must do a lot of tedious debugging before we even know where the in the code the problem occurred. If your application is complex and your use of classes extensive, you'll find this practice can save a lot of time. Also, you may want to consider creating a database specifically for logging errors.

```
    Public Function GetItem(key As String) As Variant
        Dim itm As ListItem
        On Error ErrListItemDoesNotExist Goto NoSuchItem
        Set GetItem = m_list(key)

OK:
        Exit Function

NoSuchItem:
        'return a value of Nothing if the key was not found
        Print "List item " & key & " not found."
        Set itm= Nothing
        Set GetItem = itm
        Resume OK

    End Function

    'see if there's an object in the list for a given key
    Public Function IsInList(key As String) As Integer
        Dim rval As Integer

        If ( Iselement(m_List(key)) ) Then
            rval = True
        Else
            rval = False
        End If

        IsInList = rval
    End Function

End Class
```

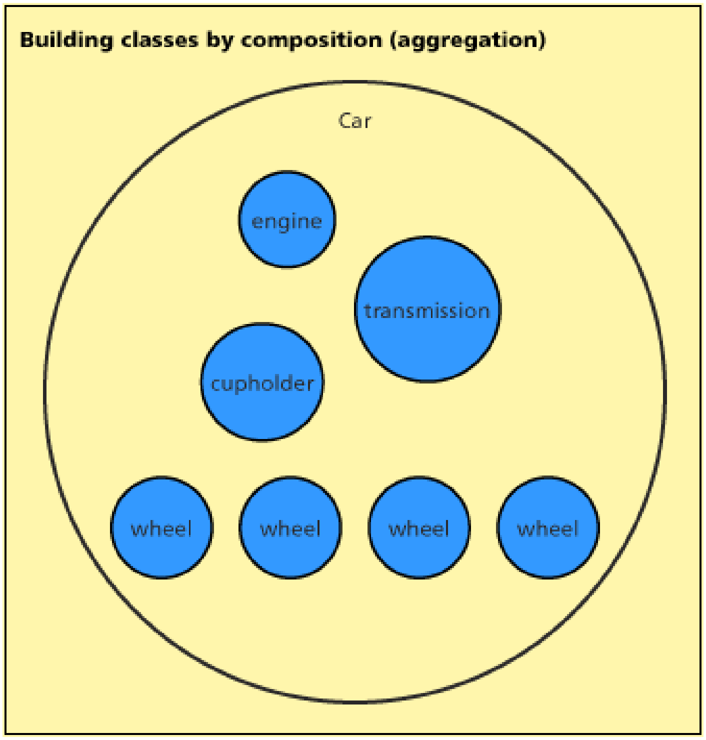
Functions and subroutines within a class (for example, GetItem or AddItem as used here) are frequently called methods in the OO world. The Sub New subroutine in a class has a special name; it's called the constructor. It gets called automatically when an object of that class is created. This subroutine is commonly used to do any initialization and setup work needed by the class.

It's also possible to have another special subroutine, the Sub Delete. It's known as the destructor. This is called automatically when an object of that class is deleted. Neither of these special subroutines is required. The constructor is so useful that I rarely leave it out. I don't bother with a destructor unless there are memory management issues to cope with. We'll discuss memory management in the **Objects and memory management** section of this article.

The theory of composition

By the way, the technical term for building a class in this way is composition—also known as aggregation. In this case, we've built a new class that contains several previously defined data types. Classes can also contain data elements that are objects themselves. We'll see how that works shortly.

In the OO world, composition or aggregation are called a "has a" relationship. A car has an engine and tires, fruit has seeds and skin, and so on. Many books explaining object-oriented languages use diagrams and code based on examples like these. Such diagrams certainly help illustrate OO concepts and below is my own version, but I'm not so sure about the code that traditionally goes with the diagrams. It works, but it doesn't do anything useful. I think the code included in this article is a better argument for using OO techniques. It shows you how to do something useful in an object-oriented way.



Creating a class by extending a built-in class

This BetterList class is useful as is; it does something that a standard Notes list can't do. We're not going to stop here though. Using the two classes we've just built, we'll build a new class that can be used to add a set of common features to all the forms in a database. We'll even be able to change those features where the standard ones are not desired. Since it's not possible to create new classes that inherit from LotusScript's built-in Notes classes, we're going to extend the capabilities of the NotesUIDocument front-end class by using the composition technique again. We'll call the new class EnhanceUIDoc.

One major benefit we'll get from this class will be the ability to detect changes made to any field in the form. Notes applications frequently do this by having an invisible "shadow" field for each field to be tracked. By using EnhancedUIDoc, we can eliminate the need to add extra fields, thus helping to keep forms simple and save developer time. If you don't want to track changes to all fields, you can create a class that inherits from EnhancedUIDoc that keeps its own list of fields to check. A profile document would be one potential place from which to load such a list. (We'll see how inheritance works shortly; in fact, we'll use EnhancedUIDoc later as the base class in our inheritance examples.)

For the sake of simplicity, this class assumes that all field values will be strings and that there will be only one value per field. It would be easy to change this class to deal with different field types and multiple values. The field types can be determined by means of NotesItem.Type property. The NotesItem.Values property returns an array that can be counted to determine if it contains a single value or multiple values.

Here's the EnhancedUIDoc class, which contains the BetterList class. (For an uninterrupted version of the code, see the **EnhancedUIDoc class** sidebar.)

```
Class EnhancedUIDoc
    Private m_uidoc As NotesUIDocument
    Private m_uiw As NotesUIWorkspace
    Private m_origvalues As BetterList
    Private m_doctype As String

    Sub ProcessPostopen( Source As NotesUIDocument )
        Dim doc As NotesDocument
        Dim ltm As ListItem

        Print ("EnhancedUIDoc - ProcessPostopen")

        Set doc = m_uidoc.document
        Forall i In doc.Items
            Set ltm = New ListItem( i.Name, i.Values(0) )
            Call m_origvalues.AddItem(i.Name, ltm)
        End Forall

      End Sub

    Sub ProcessQuerysave(Source As Notesuidocument, Continue As
    Variant)
        Dim doc As NotesDocument
        Dim ltm As ListItem
        Dim rval As Integer
        Dim v As Variant

        Print ("EnhancedUIDoc - ProcessQuerysave")
        rval = continue

        Set doc = m_uidoc.document
        Forall i In doc.Items
            Set v = m_origvalues.GetItem(i.Name)
```

Here's where we turn the variant back into an object. We must make sure it's not null first. Otherwise, Itm might not be a real object.

```
                If (Not Isnull(v) ) Then  'make sure there's an item to
                compare it to
                        Set ltm = v
                        If i.Values(0) <> ltm.value Then
                                Print "Item " & i.Name & " new value = " &
                                i.Values(0)
                                rval = True
                        Else
                                Print "Item " & i.Name & " not changed."
                        End If
                Else
                        Print "Item " & i.Name & " not found."
                End If

        End Forall

        Continue = rval
End Sub

Sub new (uid As NotesUIDocument)
        Print ("EnhancedUIDoc - sub new")

        Set m_uiw = g_wks
        Set m_origvalues = New BetterList
        Set m_uidoc = uid
```

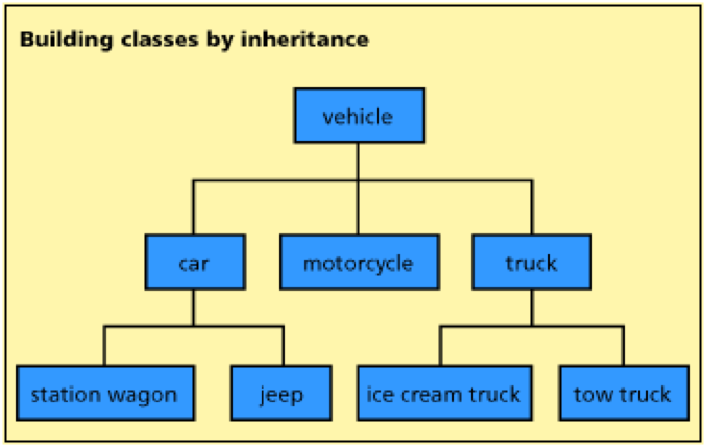
In the next lines, take a look at the On Event statements. What’s going on here? We’re replacing some generic event handling subroutines with new ones defined within our class. This is one more way that we can consolidate our code when using classes. The code in ProcessQuerysave and ProcessPostopen could have been put in the form’s Querysave and Postopen subroutines, but that would mean separating the code for these events from the code for the rest of the class.

```
        On Event Querysave From m_uidoc Call ProcessQuerysave
        On Event Postopen From m_uidoc Call ProcessPostopen

End Sub

Sub Postopen(Source As Notesuidocument)
End Sub

Sub Querysave(Source As Notesuidocument, Continue As Variant)
End Sub

End Class
```

Although this class is created using composition, it also forms the basis for our inheritance example because the next two classes we create will inherit from it.

The theory of inheritance

Before we deal with the nuts and bolts of inheritance, let’s take a brief look at the theory first. In OO books, inheritance is often called an “is a” relationship. A car is type of vehicle, a Porsche is a type of car, and so forth. Bananas, apples, and cherries are all types of fruit. Here’s another traditional OO diagram.



Subclasses are classes that inherit from another class. Subclasses are also known as derived classes or child classes. A subclass may inherit from another subclass. In the diagram above, jeep and tow truck are both subclasses that inherit from another subclass. Base classes are simply those classes that don't inherit from other classes. ListItem, BetterList, and EnhancedUIDoc are all base classes. It makes no difference that BetterList and EnhancedUIDoc make use of composition and thus contain other classes. Neither inherits from another class and that makes them base classes. In the diagram, vehicle is the only base class.

If you're like me, you're probably nodding your head at this point and saying, "Well, this inheritance stuff sounds impressive, but what can I do with it?" With your code organized into classes, you can easily create new classes based on existing ones. The only code you'll need to write for the new class is that which adds the capability the parent class lacks. You'll make future development faster and easier. Plus, you'll eliminate situations where code might be duplicated. Now let's look at how to put theory into practice.

Creating classes using inheritance

We could simply put only the capabilities we want in every form into EnhancedUIDoc and leave it at that. We've derived significant benefits already. We can do even better though. By making use of inheritance, we can quickly create new classes that can enhance, change, or override the behavior of existing classes.

One way to do this is to use EnhancedUIDoc as an abstract base class. Abstract classes don't exist to be created (or, in the more formal OO lingo, "instantiated"). They only exist to be used as a base class from which other classes will inherit. There's no simple way to force a class to be abstract in LotusScript (that is, we can't prevent someone from using it directly), but we don't really need to force it. We could use EnhancedUIDoc as the object created when creating a form, and that would work just fine; but we can choose to treat it as abstract, and I believe there is an advantage to that. I find that using one abstract base class plus a specific subclass for each form that needs to inherit common behavior is a more flexible scheme. By doing this, any code customized for one form is isolated in a subclass associated with that form. Similarly, the code intended for all forms is isolated in the base class.

Now we have a class that extends NotesUIDocument. How and where do we create this class so that it points to correct UI document? Remember, this needs to be done before the various startup events occur so that we can override these events when they do occur. We do this in the form's Queryopen event. It's not possible to do this any earlier since the

NotesUIWorkspace.CurrentDocument property doesn't return a useable value before this event.

```
Sub Queryopen(Source As Notesuidocument, Mode As Integer, Isnewdoc
As Variant, Continue As Variant)

    Set g_CurrUIDoc = New EnhancedUIDoc(Source)

End Sub
```

The global variable g_CurrrUIDoc is being used to keep track of the object belonging to the currently visible document window.

Here are two classes that inherit from EnhancedUIDoc. In the first line, we declare that class SubClassUIDoc inherits from EnhancedUIDoc by using the As keyword. The class listed after the As is the class from which new class will inherit. These are both subclasses of EnhancedUIDoc since they both inherit from it. One of them, SubclassUIDoc, changes nothing though it does contain a print statement to help illustrate the order in which methods are called. The other, NewEventUIDoc, contains code that will give it features not found in EnhancedUIDoc. Documents created using NewEventUIDoc will appear with rulers and horizontal scroll bars showing. These new features are added in the ProcessSpecialPostopen method. NewEventUIDoc also turns off the field tracking capabilities we put in its base class, NotesUIDocument. It does this by means of the "on event … remove" statement.

```
Class SubclassUIdoc As EnhancedUIDoc

    Sub new (uid As NotesUIDocument)
        Print ("SubclassUIDoc - sub new")
    End Sub

End Class

Class NewEventUIDoc As EnhancedUIDoc

    Sub new (uid As NotesUIDocument)
        Print ("NewEventUIDoc - sub new")
        On Event QuerySave From m_uidoc Remove
        On Event Querysave From m_uidoc Call
        ProcessSpecialQuerysave
        On Event Postopen From m_uidoc Call ProcessSpecialPostopen

    End Sub

    Sub ProcessSpecialQuerySave (Source As Notesuidocument,
    Continue As Variant)

        Print ("NewEventUIDoc - ProcessSpecialQuerySave")
    End Sub

    Sub ProcessSpecialPostOpen( Source As NotesUIDocument )
        m_uidoc.HorzScrollBar = True
        m_uidoc.Ruler = True
    End Sub

End Class
```

Note the "On Event QuerySave From m_uidoc Remove" statement. Without that line, we would see a call to ProcessSpecialQuerySave and ProcessQuerysave prior to saving an object of class NewEventUIDoc. The LotusScript documentation tells us that the order of calling multiple event subroutines for an object is undefined. There’s no way to know with certainty which one will be called first. Be careful not to write code that assumes that one subroutine for a particular event will be called before another. In this example, we’ve avoided the problem altogether by turning off the original subroutine.

Now that we’ve created these two subclasses, we have to consider a new issue. What happens with the constructors in the subclass and the base class when a subclass object is created? When we create a subclass object, making a call to the Sub New of the subclass also causes a call to be made to the Sub New of the class it inherits from. In other words, the base class Sub New gets called first, followed by subclass’s Sub New. The reverse happens when a destructor is called. The calls are made first to the destructor of the subclass and then on down to the base class destructor. There are print statements in the code to help illustrate this. To see this, you can download the sample database from the **Iris Sandbox**, create a SubclassUIDoc document, and observe the order in which the constructors are called.

Constructors and destructors are the only class methods that automatically call methods of the classes they inherit from. Constructors may take arguments. If they do, there are two possible subclass situations. If the subclass arguments match the parent class arguments exactly, all is well. If they don’t match, the subclass constructor must declare which arguments will be passed to the parent’s constructor. See the Property and Method Overriding topic of the **Domino R5 Designer Help** for a more complete description on how to do this. Destructors do not take arguments

Overriding properties and methods

Subclasses can override the properties and methods of their parent classes. This can be done by defining a property or method in the subclass with the same name as the one in the parent class. The parameter lists must be identical for both items. Why would we want to do this? One example might be a base class and a subclass that both have a RestoreDefaultValues method. Each class needs this capability, but each has a different set of default values to restore. What if a subclass needs to do something that its base class can already do? We already know that duplicating the code is a bad idea. A subclass can call any of the overridden methods or properties of any of the classes it inherits from by using dotdot (..) notation. Dotdot notation is only valid within classes. See the **Domino R5 Designer Help** entry on dotdot notation for a more complete description on how to do this.

Objects and memory management

Though LotusScript handles most memory management issues for you, you’ll need to pay more attention to it when you deal with objects, especially if you store objects in lists. Why is this? Every object you create takes a certain amount of memory. If you create a large number of objects without ever getting rid of ones you’re done with, your computer will bog down and may even crash. An object created within the scope of one function would get cleaned up automatically when the function ended, but an object placed in a list will not be freed until the list itself is freed up. This might not be till the database in question is closed, if there are any global variables referring to the list object.

Given that there are restrictions on the number of documents that can be open at once and that there are usually fewer than several hundred fields in a document, we’re not likely to run into this problem with EnhancedUIDoc. On the other hand, an application that tried to create a complex object for each document in a database and store each of these objects in a list might very well run low on memory when the number of documents in the database exceeded the computer’s ability to store objects representing them.

How do we avoid this problem? Be sure to delete objects when you're done with them. Also, design your code so that you don't need an excessive number of objects available at one time. The Erase statement can be used to remove all objects from a list; it can also be used to remove just one object from the list.

Limitations

If you're going to start using objects yourself, there are a few tips you should know before you begin. Objects must be defined in the declarations section of a script, and there is a 64k limit on the amount of code that can be placed in a section. If you create lots of classes, they may not all fit into one script library. If you see this message "Current operation aborted because buffer is full" when working on or when saving your script library, you've reached the 64k limit. Consider splitting an oversize script library into multiple script libraries with related classes grouped together.

At times, working with events can be tricky. Not all built-in Notes objects are available in all events. In particular, you won't be able to access a NotesUIDocument object before the form's Queryopen event. Also, the Notes back-end document for a document being created, NotesUIDocument.Document, is not available until the Postopen event.

Also, keep in mind that LotusScript compiles differently in R4 than in R5. In R4, if you try to save a script library that defines a class A and class B where class A contains an object of class B and class A is defined before class B, you'll get the error message "Class or type name not found." The same code will save correctly in R5. If you're developing for both versions of Notes, be aware of this situation.

Finally, a word of warning: Don't rush out and put all of your code into classes just to be able to flaunt the phrase *object-oriented*. Remember that just because code is object-oriented doesn't make it good code. It's perfectly possible to write bad OO code. In particular, it can be difficult to figure out code that uses more than three or four levels of inheritance. Do make use of classes when and where they make sense. As with any code, good comments and documentation will make your OO code easier to use and maintain.

Conclusion

We've examined five LotusScript classes ranging from simple to more complex. They were built using two main techniques: composition and inheritance. We've seen how composition can be used to extend the built-in Notes objects, which can't be extended by inheritance. We've used inheritance to share a basic set of behaviors between forms while allowing for specific enhancements and exceptions. We've also seen how to override events and reviewed error handling, memory management, and limitations on classes in LotusScript.

Taken altogether, we've demonstrated how LotusScript's object-oriented features can add new capabilities to LotusScript and how classes can help to avoid duplicated code. I hope the examples given here will inspire you to create some useful classes of your own.

Additional resources

Code
You can download a database containing the code discussed in this article from the **Iris Sandbox**.

Domino R5 Designer Help
Domino R5 Designer Help has some useful entries on objects. You have

to know where to look though. Try the "User-Defined Data Types and Classes" subsection of the "LotusScript Language" topic as a starting point.

You also may find the following topics of the "Property and method overriding" topic of particular interest:

- "Extending Sub New for derived classes"
- "Calling Sub New and Sub Delete"
- "Accessing base-class properties and methods"
- "Using object references as arguments and return values"
- "Using the Set statement with derived class objects"

Books
Probably the two best-known books dealing with object-oriented programming are *Object-Oriented Analysis and Design With Applications***,** by Grady Booch, Addison-Wesley, 1994, and *Design Patterns: Elements of Reusable Object-Oriented Software*, by Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995. The *Design Patterns* book looks at commonly used combinations of classes.

Web sites
Patterns are frequently used combinations of classes. Here are two Web sites that can help you learn more about them. The **Patterns FAQ** Web site includes answers to frequently asked questions. The **Patterns Home Page** Web site includes tutorials.

ACKNOWLEDGEMENTS
I'm grateful to Richard Schwartz for numerous suggestions on organizing and improving the article. I'd also like to thank Nik Shenoy for his permission to base some of my work on his. The UIDocument class he described in the 1999 Lotus Business Partner Tech Forum database inspired EnhancedUIDoc.

ABOUT BRUCE PERRY
Bruce is an independent consultant, currently working with RHS Consulting on some joint projects involving LotusScript and C++ coding. He has been working with Notes for three years. Much of that time, he worked for eVelocity Corp., a B2B Application Service Provider, where they use object-oriented LotusScript classes to work with large Notes databases—some with over one million documents. He's been a software developer since 1982 and has worked with a variety of object-oriented languages such as C++, Java, and Borland Delphi.

]

The BetterList class

```
Public Class BetterList
      Private m_list  List As Variant
      Private m_count As Integer

      Property Get Count As Integer
            Count = m_count
      End Property

      Public Function DeleteList
            Erase m_list
      End Function

      Public Sub new
            m_count = 0
      End Sub

      Sub Delete
            Call Me.DeleteList
      End Sub

      Public Function DeleteItem(key As String) As Integer
            Dim rval As Integer

            'if the key is in the list, erase the object
            If ( Iselement(m_list(key)) ) Then
                  Erase m_list(key)
                  m_count = m_count-1
                  rval = True
            Else
                  'if there's no such key, warn of an error
                  rval = False
                  Print "Item " & key & " not found.  It could not be deleted."
            End If

            DeleteItem = rval
      End Function

      Public Function AddItem(key As String, item As ListItem) As Integer

            'just add the item if the key doesn't exist
            If ( Not Iselement(m_List(key)) ) Then
                  Set m_list(key) = item
                  m_count = m_count+1
            Else
                  'if the key does exist, erase the current object in the list
                  'and add the new one
                  Erase m_List(key)
                  Set m_list(key) = item
```

```
		End If

	End Function

	Public Function GetItem(key As String) As Variant
		Dim itm As ListItem
		On Error ErrListItemDoesNotExist Goto NoSuchItem
		Set GetItem = m_list(key)

OK:
		Exit Function

NoSuchItem:
		'return a value of Nothing if the key was not found
		Print "List item " & key & " not found."
		Set itm= Nothing
		Set GetItem = itm
		Resume OK

	End Function

	'see if there's an object in the list for a given key
	Public Function IsInList(key As String) As  Integer
		Dim rval As Integer

		If ( Iselement(m_List(key)) ) Then
			rval = True
		Else
			rval = False
		End If

		IsInList = rval
	End Function

End Class
```

]

The EnhancedUIDoc class

```
Class EnhancedUIDoc
    Private m_uidoc As NotesUIDocument
    Private m_uiw As NotesUIWorkspace
    Private m_origvalues As BetterList
    Private m_doctype As String

    Sub ProcessPostopen( Source As NotesUIDocument )
        Dim doc As NotesDocument
        Dim ltm As ListItem

        Print ("EnhancedUIDoc - ProcessPostopen")

        Set doc = m_uidoc.document
        Forall i In doc.Items
            Set ltm = New ListItem( i.Name, i.Values(0) )
            Call m_origvalues.AddItem(i.Name, ltm)
        End Forall

    End Sub

    Sub ProcessQuerysave(Source As Notesuidocument, Continue As Variant)
        Dim doc As NotesDocument
        Dim ltm As ListItem
        Dim rval As Integer
        Dim v As Variant

        Print ("EnhancedUIDoc - ProcessQuerysave")
        rval = continue

        Set doc = m_uidoc.document
        Forall i In doc.Items
            Set v = m_origvalues.GetItem(i.Name)

            If (Not  Isnull(v) ) Then   'make sure there's an item to compare it to
                Set ltm = v
                If i.Values(0) <> ltm.value Then
                    Print "Item " & i.Name & " new value = " & i.Values(0)
                    rval = True
                Else
                    Print "Item " & i.Name & " not changed."
                End If
            Else
                    Print "Item " & i.Name & " not found."
            End If

        End Forall

        Continue = rval
    End Sub

    Sub new (uid As NotesUIDocument)
        Print ("EnhancedUIDoc - sub new")

        Set m_uiw = g_wks
        Set m_origvalues = New BetterList
        Set m_uidoc = uid

        On Event Querysave From m_uidoc Call ProcessQuerysave
        On Event Postopen From m_uidoc Call ProcessPostopen

    End Sub

    Sub Postopen(Source As Notesuidocument)
    End Sub

    Sub Querysave(Source As Notesuidocument, Continue As Variant)
    End Sub

End Class
```