The Raiser's Edge® Enterprise™

API Essentials Guide

022708

RE7Enterprise-APIGuide-022708

API Essentials Guide

Contents

What Is In This Guide?

Using the *API Essentials Guide*, your technical staff can learn how to use the optional module *API for Advanced Application Development* to customize programs that integrate with your ***Raiser's Edge*** system. For example, a program developer can create seamless links from ***The Raiser's Edge*** to other software programs, such as patient tracking, ticketing, and the Internet. You can also learn about the following:

- "The Raiser's Edge Object Fundamentals" on page 11
- "Advanced Concepts and Interfaces" on page 79
- "Custom View: Creating Custom Parts" on page 84
- "API Programming Fundamentals" on page 97
- "The Raiser's Edge Object MetaViewer" on page 125
- "API Code Samples" on page 126
- "Plug-In Code Samples" on page 127

How Do I Use These Guides?

The Raiser's Edge user guides contain examples, scenarios, procedures, graphics, and conceptual information. Side margins contain notes, tips, warnings, and space for you to write your own notes.

To find help quickly and easily, you can access the ***Raiser's Edge*** documentation from several places.

User Guides. You can access PDF versions of the guides by selecting **Help**, **User Guides** from the shell menu bar or by clicking **Help** on the Raiser's Edge bar in the program. You can also access the guides on our Web site at www.blackbaud.com. From the menu bar, select **Support**, **User Guides**.

In a PDF, page numbers in the Table of Contents, Index, and all cross-references are hyperlinks. For example, click the page number by any heading or procedure on a Table of Contents page to go directly to that page.

Help File. In addition to user guides, you can learn about ***The Raiser's Edge*** by accessing the help file in the program. Select **Help**, **The Raiser's Edge Help Topics** from the shell menu bar or press **F1** on your keyboard from anywhere in the program.

Narrow your search in the help file by enclosing your search in quotation marks on the Search tab. For example, instead of entering Load Defaults, enter "Load Defaults". The help file searches for the complete phrase in quotes instead of individual words.

Essentials

Chapter 1

Contents

This guide provides Visual Basic developers with all the information needed to customize and enhance ***The Raiser's Edge***. From a quick *VBA* macro to a full blown application based on ***The Raiser's Edge*** Object *API*, you can find what you need here. A wealth of programming samples are provided to illustrate key concepts and provide you with a solid foundation on which to build your custom ***Raiser's Edge*** solutions.

Please remember...

We provide programming examples for illustration only, without warranty either expressed or implied, including, but not limited to, the implied warranties of merchantability and/or fitness for a particular purpose. This guide assumes you are familiar with Microsoft *Visual Basic* and the tools used to create and debug procedures. Blackbaud Customer Support can help explain the functionality of a particular procedure, but they will not modify, or assist you with modifying, these examples to provide additional functionality. If you are interested in learning more about ***The Raiser's Edge*** *VBA* and *API* optional modules, contact our Sales department at solutions@blackbaud.com.

The programming examples and related code provided to you via this guide are the property of Blackbaud, Inc. and you may not copy, distribute, convey, license, sublicense, or transfer any rights therein. All examples are subject to applicable copyright laws.

We hope you find this guide useful as you develop for ***The Raiser's Edge***. If you are not sure if this material is targeted for you, see "Using This Guide" on page 3.

If you have programmed in *Visual Basic* before, we suggest you review the "Documentation Map" on page 4. This map is a great starting point from which you can navigate to the particular aspect of ***Raiser's Edge*** programming that interests you.

Using This Guide

This guide is for developers who are creating solutions for ***The Raiser's Edge***. These solutions range from creating a basic *VBA* procedure to large, complex addition to ***The Raiser's Edge***.

The information is laid out in a clean, progressive fashion. We introduce concepts and techniques gradually. If you are familiar with *Visual Basic* programming and object-oriented programming concepts, the content will be useful. *VBA* programming in ***The Raiser's Edge*** should be very familiar to anyone who has written either *Visual Basic* code or *VBA* code in other applications (Microsoft *Office*, for example). *API for Advanced Application Development* should be easy for any developer who has used COM objects from *Visual Basic*.

This information is not for everyone. To cover the material in a useful manner, we had to make certain assumptions about your level of knowledge. If you are comfortable with the Visual Basic language and you understand data types, variable scoping, and how to use the *Visual Basic* Editor, then this guide is for you. If not, your time may be better spent with one of the many fine introductory materials available. Actually, one of the best resources is the online help provided with *VBA*.

For more information...

Visit Blackbaud's Web site at www.blackbaud.com for software customization FAQs, code samples, and other helpful information, such as error explanations. The VBA\API Web site page is one of your primary sources of information for customizing your ***Raiser's Edge*** software. You can also send an email to dssupport@blackbaud.com or call 1-800-468-8996 for support assistance.

Documentation Map

This guide is broken down into logical sections, each designed to help you learn and use a particular aspect of the available extensibility technologies. Because there is important information that applies to both *VBA for Advanced Customization* and *API for Advanced Application Development* optional modules, some documentation for both products is included in the Essentials chapter. If you come across *VBA* or *API* information that is not applicable to your organization's ***Raiser's Edge*** software package, contact Sales at solutions@blackbaud.com for more information.

The Essentials

This chapter introduces key concepts that you need to understand as you program with ***The Raiser's Edge***.

Visual Basic for Applications (VBA)

This chapter details *VBA* support in ***The Raiser's Edge***. *VBA* is an extremely powerful, industry-standard programming environment built right into ***The Raiser's Edge***.

Application Program Interface (API)

This chapter exposes core functionality using *API*, enabling developers to build custom solutions that leverage ***Raiser's Edge*** technology through a set of easy to use COM objects.

Programmer's Reference

This section, located only in the help file, provides a detailed listing of all objects and services available to developers programming with ***The Raiser's Edge*** object model.

Programming Language

The code samples in this guide are written using *Visual Basic 6.0* language. This language is shared by *VBA*, Microsoft *Visual Basic 6.0*, Microsoft *Office 2000*, and other *VBA* 6.0 host applications. While it is possible to use *API* from other languages (C++ or Java, for example), Blackbaud can only provide support regarding *Visual Basic* programming.

Sample Code

Periodically, we provide code samples to illustrate a point. Code samples always appear in the following format.

```
'Programming Example -
    '    we will put VB code comments in Green

    Dim oGift as CGift
    Set oGift = New CGift

    oGift.Init REApplication.SessionContext
```

Note....

You may notice occasional line breaks in the code due to layout constraints. All line breaks are documented with the standard " _ " in the code sample.

Note that we sometimes clarify points in the code samples using standard *Visual Basic* comments. To accentuate them, all comments appear in green.

Raiser’s Edge Programming Essentials

This section covers the items that make the foundation of ***Raiser’s Edge*** programming. We introduce the terms and skills you need to use ***Raiser’s Edge*** objects, and we take you step-by-step through several simple examples.

Objects and Object Models

This section provides a general overview of COM automation objects and object models.

The Raiser’s Edge Type Library

The Raiser’s Edge Type Library provides “early bound” access to the system’s objects and functions with any COM compatible language.

The SessionContext

This section introduces the most important object in the system. No Blackbaud programming task can be tackled without using a SessionContext.

Initializing and Releasing Objects

This section outlines the mechanics of creating and destroying objects, as almost every object in ***The Raiser’s Edge*** must be initialized and released in the same fashion.

Data Objects

All data elements in ***The Raiser’s Edge*** are modeled using data objects. Data objects provide a high-level COM programming interface to every data record in ***The Raiser’s Edge***.

User Interface (UI) Objects

User interface objects allow for programmatic access to many of the forms and windows that comprise ***The Raiser’s Edge*** user interface.

Service Objects

These objects provide a high-level interface to the system level functionality in various ***Raiser’s Edge*** modules such as *Query*, *Export*, and *Reports*.

Advanced Concepts and Interfaces

This section discusses some advanced topics such as Interfaces and Transactions, that are available in ***The Raiser’s Edge*** object model.

Objects and Object Models

The Raiser's Edge was built from the ground up using objects. Nearly everything you do in *Visual Basic* involves manipulating objects.

Please remember...

Every ***Raiser's Edge*** data element—such as each constituent, gift, campaign—is an object that you can manipulate programmatically in *Visual Basic*.

Every data element — each constituent, gift, campaign, and so on — is an object that you can manipulate programmatically in *Visual Basic*. Once you understand how to work with objects, you are ready to program ***The Raiser's Edge***.

What Are Objects and Object Models?

The Raiser's Edge consists of two things: content and functionality. Content refers to the data elements the system contains: the constituents, gifts, contacts, campaigns, funds, and appeals. Content also refers to information about attributes of individual elements in the application, such as the amount of a gift or the number of registrants for an event. Functionality refers to all the ways you can work with the content in ***The Raiser's Edge*** — for example, opening, closing, adding, or deleting records in the application.

Content and functionality are broken down into discrete, related units called objects. You are probably already familiar with some of these objects, as elements of ***The Raiser's Edge*** user interface. One example is the constituent record, which is presented as one window with many tabs allowing access to subsets of the constituent's object model.

To become a productive ***Raiser's Edge*** programmer, it is important to understand how the object model is organized. We built this model with the goal of providing consistent, hierarchical access to all the data elements in the system. Blackbaud's development team used these very same objects to build ***The Raiser's Edge***!

The Raiser's Edge object model has one major goal: to expose all important functionality and data needed to manipulate the database records and services in a high-level manner.

The Raiser's Edge Object Model

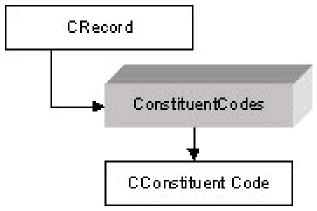
The Raiser's Edge object model is best described as a group of object models. Each major record type in the system has its own hierarchical object model. For example, the Gift has a large model comprised of "child" objects such as installments, split funds, and attributes. The concept of a child object is an important one to grasp. By child object, we mean an object that is only accessible via an object that is above it in the object model. You cannot create child objects as free standing objects; they must be created via a method on their parent's object. By building these models to reflect the layout of their real-world counterparts, the task of programming an extremely large, complicated relational database such as ***The Raiser's Edge*** is simplified to a manageable level.

In addition to containing lower-level objects, each object in the hierarchy contains content that applies to both the object itself and all objects below it.

Data Objects

The Raiser's Edge object model is based primarily around the data that the program manages. It does not expose the interface as a programmable entity. Because the key to your ***Raiser's Edge*** system is the data that it manages, data objects are the key to programming ***The Raiser's Edge***.

Let’s take a look at a simple example. In ***The Raiser’s Edge***, constituents can be assigned a constituent code. Constituent codes are used to denote important information on each record (such as Alumni, Past Parent, Major Donor). A constituent can have any number of constituent codes on her record (in relational database terms there is said to be a “one-to-many” relationship between a record and its constituent codes). A constituent code has no meaning outside of a record. For this reason, in ***The Raiser’s Edge*** data object model, the ConstituentCodes object is a child of the CRecord data object, which is the object in the system that represents constituents.



In this diagram, we see that for each CRecord there is a child object named ConstituentCodes, and the ConstituentCodes object has child object named CConstituentCode. The ConstituentCodes object name is plural for a very important reason. It is a collection object. This means it is used to iterate through any number of children (in this case constituent codes). All collection objects in ***The Raiser’s Edge*** object model can be navigated using “For Each” syntax, which is the standard for navigating *VBA* collections. Take a look at the next code sample. Don't worry about the details—they are introduced later in this guide.

```
'Note: The code to initialize and load a CRecord (oRecord)
'    object omitted for brevity

    Dim oConsCode as CConstituentCode

    'Print all of this constituent's constit codes to the
     '    VBA debug window
    For Each oConsCode in oRecord.ConstituentCodes
        Debug.Print oConsCode.Fields(CONSTITUENT_CODE_fld_CODE)
    Next oConsCode
```

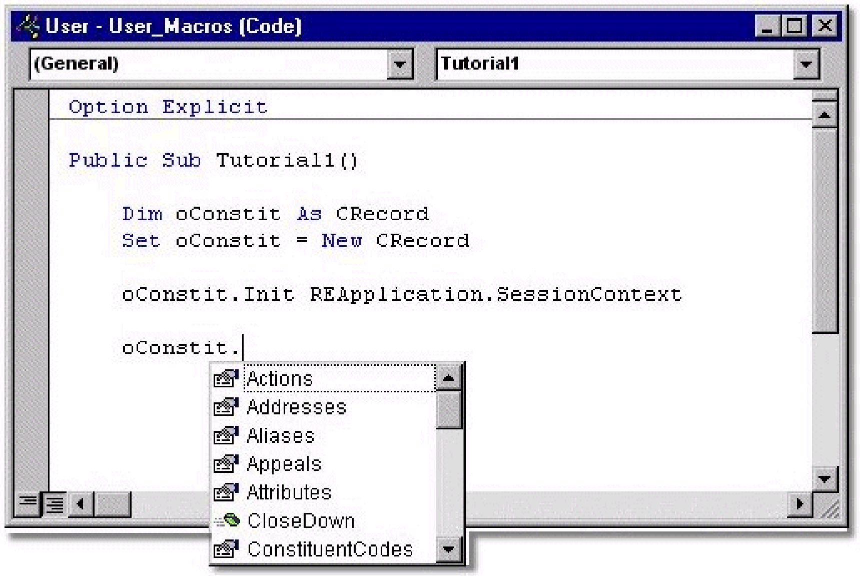
The Raiser’s Edge Type Library

The easiest and most efficient way to use ***Raiser’s Edge*** objects is through the provided type library. A type library is a language independent file that provides type information about the components, interfaces, methods, constants, enumerations, and properties exposed by ***The Raiser’s Edge***. While this online guide discusses only Visual Basic, type libraries can be used from any language (for example, in Microsoft *Visual C++*, the developer can use the #import statement).

Using Early Bound Objects and the Type Library

Without using a type library, *Visual Basic* is limited to communicating to components through the dispatch interface, which is slow and provides no compile-time syntax checking. Once you have a reference to ***The Raiser’s Edge*** Type Library, you can use the object browser provided by Visual Basic to explore the objects, and can easily get help on any property or method.

Another incredible productivity gain that becomes available when using type libraries with *Visual Basic 5.0* and higher (or *VBA*) is Intellisense. If you have worked with *VB* or *VBA*, you have probably noticed while programming with objects that when you hit "." after an object variable defined as a type described in a type library (or Visual Basic component) the code editor appears and displays a list similar to the one in the following graphic.



```
Option Explicit

Public Sub Tutorial1()

    Dim oConstit As CRecord
    Set oConstit = New CRecord

    oConstit.Init REApplication.SessionContext

    oConstit.
```

VB's Intellisense feature displays only the properties and methods that are available on the object. In the above graphic, you see properties and methods such as Actions, Addresses, Aliases. These are all child objects that are available for a CRecord object. VB can only work this magic if you are using an early-bound object. By early-bound, we mean an object variable that is declared as a specific type. Take a look at the following code sample.

```
'This variable is late bound.  While it will still work,
'    it will incur significant runtime overhead, and it will
'    yield no intellisense

Dim oRecord As Object
Set oRecord = New CRecord

'This early-bound variable provides optimal speed and
'    access to the VB/VBA intellisense feature.

Dim oRecordEarly As CRecord
Set oRecordEarly = New CRecord
```

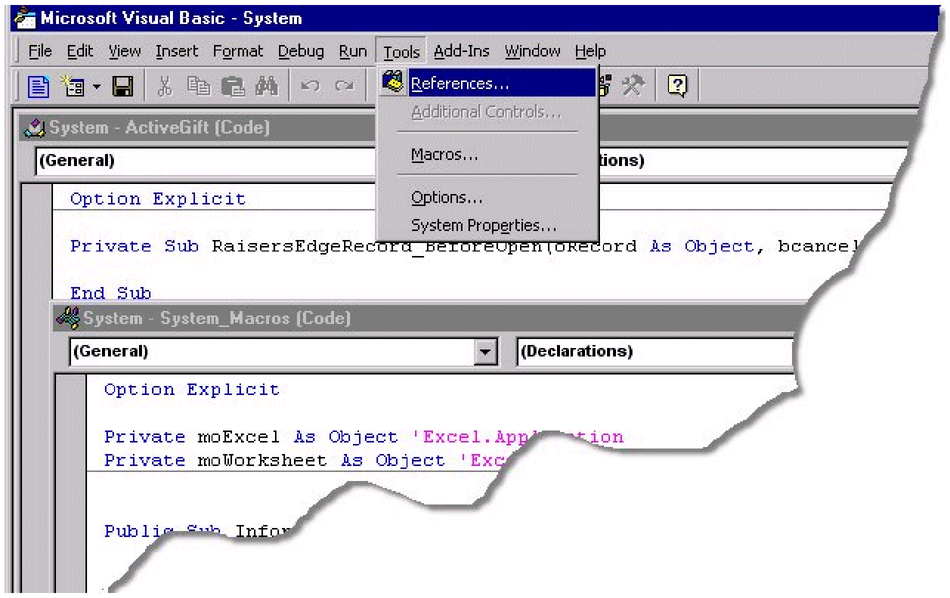
Using the Type Library from VBA

When you have the optional module *VBA*, the system automatically sets a reference to ***The Raiser's Edge*** Type Library when you start *VBA*. Each of the two provided *VBA* projects has a reference to the library.

For more information about *VBA*, see the VBA chapter.

Accessing the References Dialog

You can manually set a reference to the library by selecting **Tools, References** from the menu bar in *VBA*.

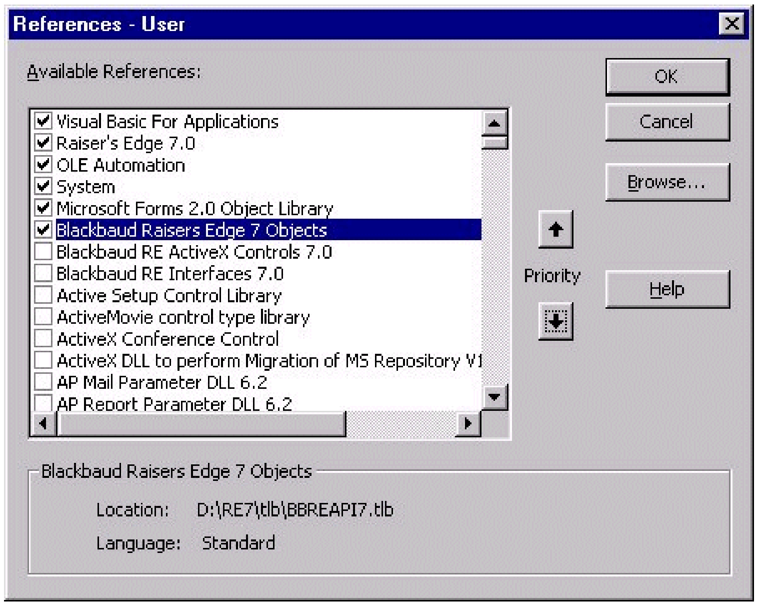

Microsoft Visual Basic - System
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
References...
Additional Controls...
Macros...
Options...
System Properties...
System - ActiveGift (Code)
(General)
tions)
System - System_Macros (Code)
(General)
(Declarations)

```
Option Explicit

Private Sub RaisersEdgeRecord_BeforeOpen(oRecord As Object, bcancel

End Sub
```

```
Option Explicit

Private moExcel As Object 'Excel.Application
Private moWorksheet As Object 'Exc


Public Sub Infor
```

Setting a Reference to The Raiser's Edge Type Library

When you select **References**, the References dialog appears with a list of various type library references already set. The most important of these (for our purposes) is Blackbaud Raisers Edge 7 Objects. This is the reference you must set to gain early-bound access to ***Raiser's Edge*** Objects.



- [✓] Visual Basic For Applications
- [✓] Raiser's Edge 7.0
- [✓] OLE Automation
- [✓] System
- [✓] Microsoft Forms 2.0 Object Library
- [✓] Blackbaud Raisers Edge 7 Objects
- [ ] Blackbaud RE ActiveX Controls 7.0
- [ ] Blackbaud RE Interfaces 7.0
- [ ] Active Setup Control Library
- [ ] ActiveMovie control type library
- [ ] ActiveX Conference Control
- [ ] ActiveX DLL to perform Migration of MS Repository V1
- [ ] AP Mail Parameter DLL 6.2
- [ ] AP Report Parameter DLL 6.2

Blackbaud Raisers Edge 7 Objects

Location: D:\RE7\tlb\BBREAPI7.tlb

Language: Standard

Note...

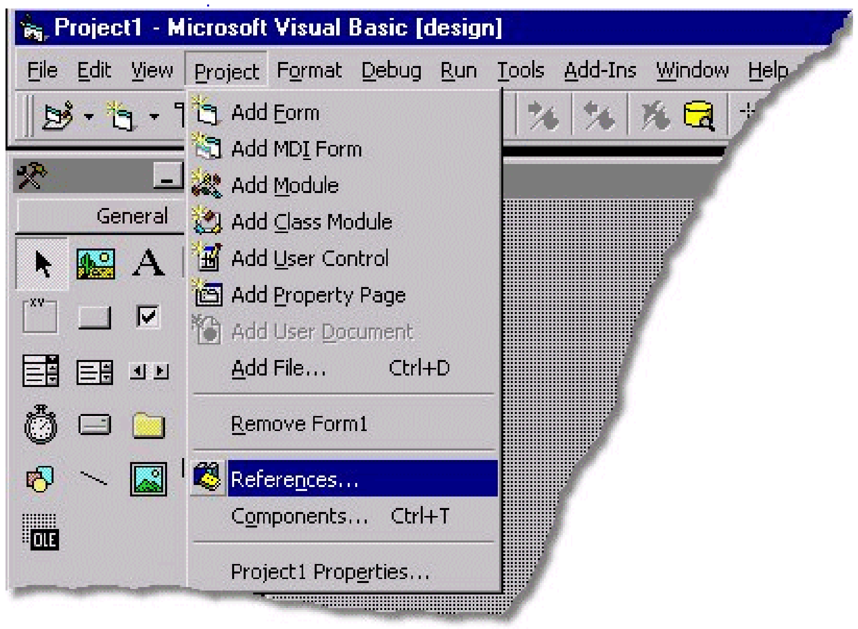
If you unmark the **System** checkbox, you must exit ***The Raiser's Edge*** and enter the program again to restore System references. System references load when you enter ***The Raiser's Edge***. Therefore, if you try to add the reference back on the References dialog, an Automation error generates.

Using the Type Library from an API Application

If you have the optional module *API*, you need to set a reference to the type library from any *Visual Basic* project that you want to gain early-bound access to ***Raiser's Edge*** objects.

Accessing the References Dialog from Visual Basic 5.0 and Higher

To set a reference to the library from *Visual Basic* 5.0 or higher, create a new VB project and select **Project, References** from the menu bar.



The Raiser's Edge Object Fundamentals

This section introduces the mechanics of using ***Raiser's Edge*** objects. First we introduce "The SessionContext" on page 11, which is the key to ***Raiser's Edge*** programming. Next, we explain some important methods involved in "Initializing and Releasing Objects" on page 13. We also break down each object type in the system and provide code samples and discuss how to use them.

The SessionContext

Whenever you use an object exposed by ***The Raiser's Edge*** object model, it must be initialized first. All objects are initialized with a very important object parameter, the SessionContext. The SessionContext is the key to programming ***The Raiser's Edge***. This object holds information regarding the state of the active instance of ***The Raiser's Edge*** application.

Please remember....

The SessionContext is the key to programming ***The Raiser's Edge***. This object holds information regarding the state of the active instance of ***The Raiser's Edge*** application

When you create new instances of objects and initialize them with a SessionContext, the object queries the SessionContext for important information they need to operate (for example, a handle to the low-level database connection interface).

Accessing the SessionContext from VBA

When the *VBA* environment is initialized, ***The Raiser's Edge*** exposes its SessionContext via the *API* object. The *API* object is a global object available to *VBA*. The most important property on the *API* object is the SessionContext. The code sample initializes a CGift object for use in *VBA*.

```
Dim oGift as CGift
    Set oGift = New CGift

    'Use the REApplication object to get a reference to the
    '     SessionContext
    oGift.Init REApplication.SessionContext

    'Load Gift 1
    oGift.Load 1

     'Release reference to Gift Object
    oGift.CloseDown
```

Accessing the Session Context from API

Like *VBA*, objects must be initialized when using *API*. It is important to understand that while a few differences exist, once you understand ***Raiser's Edge*** object programming, the same rules apply to both *VBA* and the *API*.

An *API* application obtains its reference to the SessionContext via the *API* object. Unlike the REApplication object, which is automatically initialized and available to *VBA* in the running instance of ***The Raiser's Edge***, *API* must be initialized by the programmer. The code sample accomplishes the same task as the previous *VBA* sample.

```
Dim oAPI as REAPI

     'Initialize the API and log in
    Set oAPI = New REAPI

     'Log in as Raiser's Edge user Judy with password "Admin"
    oAPI.Init "Judy", "Admin"

    Dim oGift as CGift
    Set oGift = New CGift

    'Use the API object to get a reference to the
     '    SessionContext
    oGift.Init oAPI.SessionContext

     'Load Gift 1
    oGift.Load 1

     'Release reference to Gift Object
    oGift.CloseDown
```

Note the similarities to the earlier *VBA* sample. The first three lines of code in the sample remain constant for any *API* application and are usually placed in a section of your *API* application that is executed only once (for example, in your main form's Load event).

Initializing and Releasing Objects

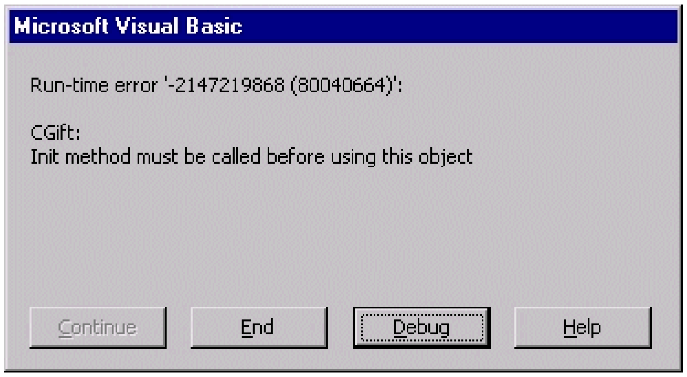
To properly initialize a ***Raiser's Edge*** object, you pass a reference to the SessionContext. Almost every top-level object in ***The Raiser's Edge*** is initialized this way.

The Init and CloseDown Methods

The *VBA* code sample is representative of almost every sample of object programming code you see in the ***The Raiser's Edge***.

```
Dim oGift as CGift
    Set oGift = New CGift

     'Initialize the oGift via the init method
    oGift.Init REApplication.SessionContext

     ' Load Gift 10
    oGift.Load 10

     ' Properly release reference to Gift Object using the CloseDown method
    oGift.CloseDown
```

Initialize (.Init) with a SessionContext and release (.CloseDown) the object when you are done. If you attempt to use a ***Raiser's Edge*** object without properly initializing it, a trappable runtime error is raised.



Microsoft Visual Basic

Run-time error '-2147219868 (80040664)':

CGift:
Init method must be called before using this object

Closing down objects can be harder. If you fail to properly CloseDown an object, potentially all the object resources remain "alive" and in memory. To many developers, this is known as a "memory leak". The objects attempt to detect this situation and raise errors in many situations if a .CloseDown call was not made. In some cases this type of leak cannot be detected immediately, leading to some hard-to-track bugs. Remember, if it has an .Init method, it probably has a .CloseDown method also, and you should always make sure you call them both.

Data Objects

Most ***Raiser's Edge*** programming involves data objects. As discussed in "Objects and Object Models" on page 6, data objects provide a high-level abstraction layer over the records in the underlying ***Raiser's Edge*** database. In this section we learn the basics of programming with data objects.

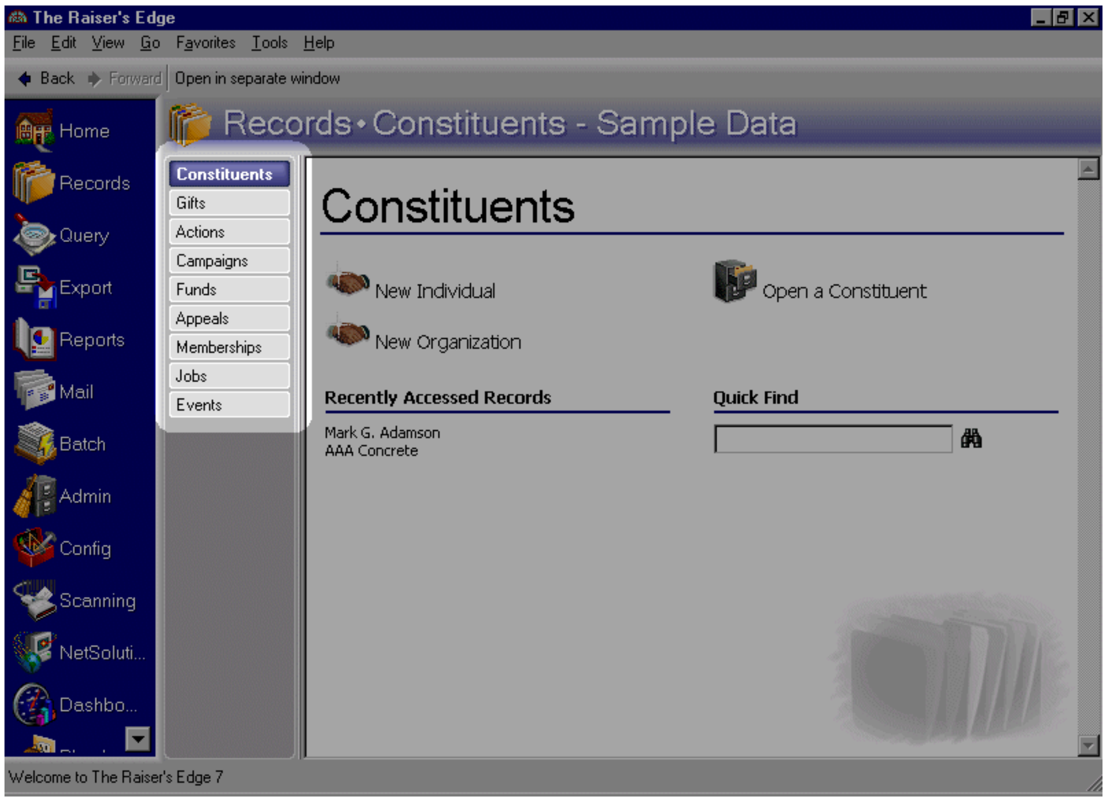
Data Object Hierarchy

To manipulate a data record in your ***Raiser's Edge*** system, you initialize and load the appropriate data object. ***The Raiser's Edge*** object model provides a data object for every editable record in your system. Only a select few data objects can be instantiated and loaded. Most data objects are "children" of another object in the hierarchy. For example, your database may have thousands of constituents who are coded with a constituent code of AL (Alumni). Therefore, in your database there are thousands of ALUMNI constituent code records stored. Alone, each of these records has little value. However, they do have meaning in the context of the specific constituent to which they are related. The constituent code object is accessed as a "child" of the constituent object (CRecord, to be exact).

What Are "Top Level" Objects?

Understanding the parent-child data object relationship is a key concept to grasp as you move forward with data object programming. Throughout this guide, you see objects at the top of the object hierarchy referred to as "Top Level Objects"; any objects that are accessible only via a top level object are referred to as "child" objects.

Which objects are parents and which are children? The easiest way to familiarize yourself with the hierarchy is to look at ***The Raiser's Edge*** application. When end-users are performing day-to-day data entry chores, they access records through the Records page in ***The Raiser's Edge*** shell.



Mark G. Adamson
AAA Concrete

The previous graphic shows the standard ***Raiser's Edge*** Records page. The highlighted buttons are top level objects. Just as the end-user must first open a Constituent to access his constituent codes, you, as the developer, must load a constituent object first before navigating to the constituent's constituent codes.

In addition to the items above, constituent relationships and event participants are also top level objects.

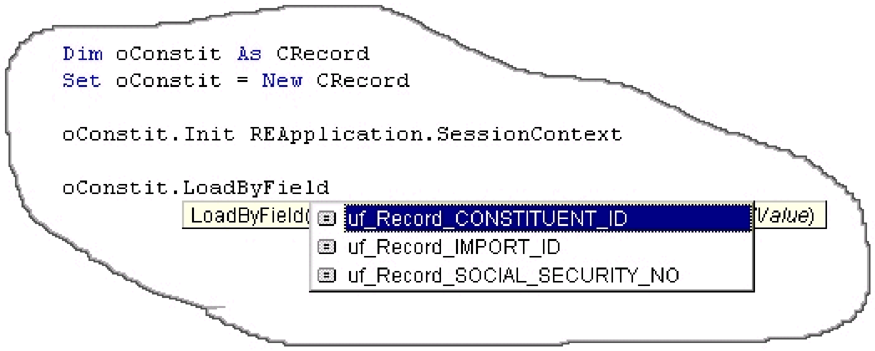
Loading a Data Object

Now that we have introduced some concepts, it is time to start programming with data objects. This section introduces some common ways to load a data object.

How Many Ways Can I Load a Data Object?

Each data object supports various methods to allow for loading. Each ***Raiser's Edge*** top-level object can be loaded using any of its unique fields. For example, ***The Raiser's Edge*** does not allow an end-user to save two constituent records with the same constituent ID, or two campaign records with the same campaign ID. Given this, you can load each data object using the underlying record's unique fields with the LoadByField method.

LoadByField accepts two arguments. The first argument denotes the unique field you use. The second argument provides the key to search for. Here, you can see another great example of how powerful *Visual Basic's* Intellisense feature can be. Because each top-level data object has a different set of unique fields, the object's corresponding LoadByField enumerates these fields in a drop-down list as you fill in the first argument to the LoadByField method.



```
Dim oConstit As CRecord
Set oConstit = New CRecord

oConstit.Init REApplication.SessionContext

oConstit.LoadByField
```

In the code snippet above, we see the programmer has instantiated a CRecord object. As mentioned earlier, CRecord is the data object that encapsulates individual and organization constituent records. Note the drop-down list that appeared automatically when the developer entered the unique field argument. *Visual Basic* "knows" which arguments are valid in the context of a CRecord because ***The Raiser's Edge*** type library exposes this information. The productivity boost gained here cannot be overstated. As you program with ***Raiser's Edge*** objects, you will see that throughout the system, arguments are exposed to Intellisense in this fashion.

An Alternate Method to Load Data Objects—The Database ID

Each record in ***The Raiser's Edge*** is stored in the database. To define database relationships and integrity, the records are assigned unique values by the Database Management System (DBMS). These values are called Primary Keys. Each top-level data object can be loaded using this key value with the "Load" method. The load method accepts just one argument, a long integer representing the primary key of the record you want to load.

Code Sample

These code samples show the various ways to load a constituent data object for the hypothetical constituent "Michael Simpson". Mr. Simpson has a social security number of 025-64-6381, and a database (primary key) of 166.

```
Dim oConstit as CRecord
Set oConstit = New CRecord

oConstit.Init REApplication.SessionContext

'Load the record via the Social Security Number
oConstit.LoadByField uf_Record_SOCIAL_SECURITY_NO, "025-64-6382"
```

```
'Load the record via the Database ID
oConstit.Load 166
```

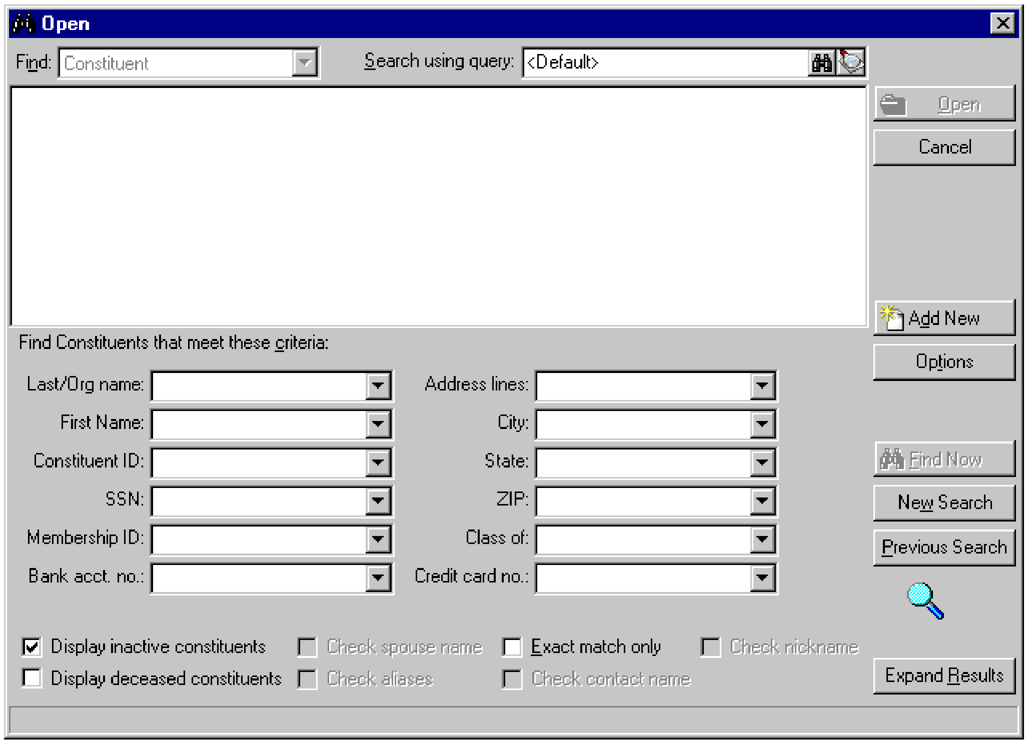
Using The Raiser's Edge Search Screen to Load Your Data Object

So far, we have seen how to load a data object given a specific search key. Many times this is a completely acceptable solution (for example, if you are building your own customized search screen). However, in some cases you may require a more robust search, or you may want to concentrate on your application and use as many pre-built components as possible.

The Raiser's Edge exposes its search screen as a programmable entity. Using the standard ***Raiser's Edge*** search screen from *Visual Basic* code is easy. ***The Raiser's Edge*** search screen is referred to as a Service Object, meaning it is an object that provides easy access to ***Raiser's Edge*** functionality. We are jumping ahead a little here—the many service objects provided by ***The Raiser's Edge*** are discussed later in this guide, but at this point it is important to at least introduce the concept of using the search screen to load a data object.

```
Dim oConstit as CRecord

     'Access the SearchScreen service object
    Dim oServices as REServices
    Set oServices = New REServices
    oServices.Init REApplication.SessionContext

     'Declare variable used to access the SearchScreen
    Dim oSearch As IBBSearchScreen

     'The services object exposes most common, useful interface dialogs
    Set oSearch = oServices.CreateServiceObject(bbsoSearchScreen)
    oSearch.Init REApplication.SessionContext

     '"Tell" the search dialog to allow for a constituent search
    oSearch.AddSearchType SEARCH_CONSTITUENT

     'Show The Search form (See Figure 4)
    oSearch.ShowSearchForm

    'If The user didn't cancel - assign the
     '    record they selected to our data object
    If Not oSearch.SelectedDataObject Is Nothing Then
        Set oConstit = oSearch.SelectedDataObject
    End If
```

The search screen (from the UI, this is the Open screen) is the end result of the previous code sample. The end-user is presented with the standard ***Raiser's Edge*** search dialog. If the end-user selects a record, the search service constructs the appropriate data object, which we access from code via the "SelectedDataObject" property.

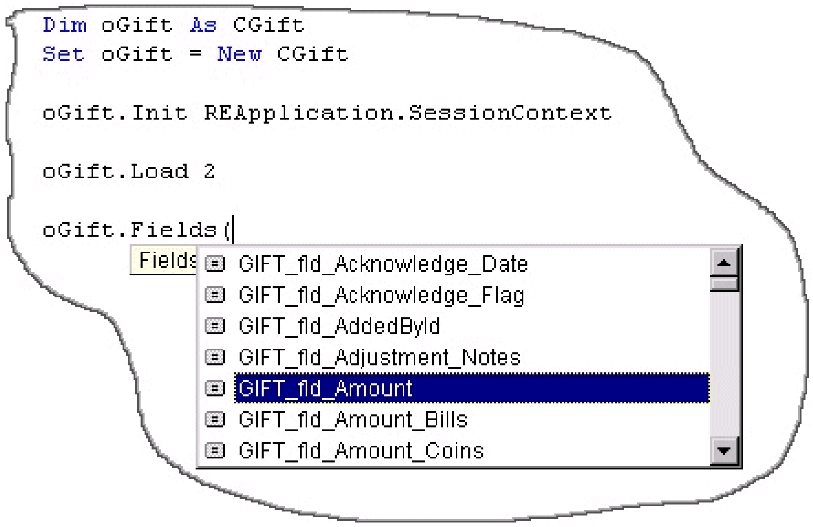


Updating Data Objects

Previously in this guide, we explored "Initializing and Releasing Objects" on page 13 and "Loading a Data Object" on page 15 of data objects. In this section, you learn how to use the data objects to update our database records.

The Fields Property

Each data object shares a common and very important property—Fields. The Fields property exposes all the individual, updatable data elements that make up a data object. Instead of exposing a unique property for each field on a data object, which would be cumbersome and very hard to extend, Blackbaud's developers built the Fields property. With the Fields property, when you access the Fields property from code, a list appears showing the constants for all valid fields on the object. This way, there is no time spent searching through hundreds of properties on an object just to find the **Name** field. This design also enables Blackbaud to easily add new fields as ***The Raiser's Edge*** evolves, without breaking any existing code. Review the code sample below.



```
Dim oGift As CGift
Set oGift = New CGift

oGift.Init REApplication.SessionContext

oGift.Load 2

oGift.Fields(
```

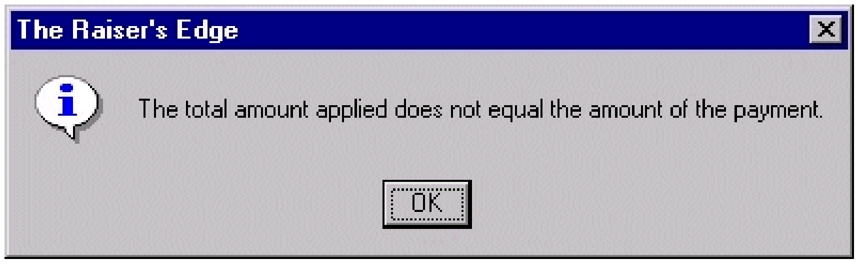
In this sample, you can see the developer has accessed the fields property of a gift data object. He is presented with a drop-down of all the available fields on a gift, and he is selecting "GIFT_fld_amt" (which represents the **Gift Amount** field on the gift record). Review the complete code sample below that loads a gift from the database into the gift data object, increments the **Gift Amount** field (by $10.00), and then saves the gift.

```
Dim oGift As CGift
    Set oGift = New CGift

    oGift.Init REApplication.SessionContext
    oGift.Load 2

    'Update the gift amount field
     '    Increment it by 10 dollars...

    oGift.Fields(GIFT_fld_Amount) = oGift.Fields(GIFT_fld_Amount) + 10

     'Save our changes
    oGift.Save

     'Clean Up
    oGift.CloseDown
    Set oGift = Nothing
```

This sample illustrates how simple it is to update your data using data objects. Remember, if we put invalid data into the amount field (for example, "xxxx") when we issue the .Save method on the object, the data object raises a trappable error.

Validation and Integrity

Data object validation goes much further than just filtering out bad data. Every Business Rule in the system is checked, both internal rules and rules your end-users established using Business Rules in ***The Raiser's Edge***. For example, in ***The Raiser's Edge***, if end-users attempt to over-apply a pledge payment, the following message appears.


The Raiser's Edge

The total amount applied does not equal the amount of the payment.

If a *Visual Basic* developer attempts to over-apply a pledge using *Visual Basic* code, a trappable error is raised with the same message (accessible via the err.description property on the Visual Basic Error object). You will learn about trapping and handling data object errors later. For now, it is important to understand this validation exists to maintain a high level of consistency and integrity in your database.

The main point to remember is the object insulates your database, and no "garbage" can make it to the database without first being validated by the object. This rule applies to every facet of the data element. ***The Raiser's Edge*** uses code just like the code provided in the examples, so you can be sure that updates using data objects are consistent with updates made by end-users in the system.

Adding and Deleting Data Objects

You can add and delete data objects with code.

Adding a Record Using a Data Object

When an end-user wants to add a new constituent record to ***The Raiser's Edge***, he clicks **Records** from the Raiser's Edge bar, and then clicks **New Individual** or **New Organization**. You can also do this through code, as the following sample illustrates.

```
'Create a new instance of the CRecord object
    Dim oRec As CRecord
    Set oRec = New CRecord

     'Initialize the object by passing in a valid SessionContext
    oRec.Init REApplication.SessionContext

     'Set any values and save
    oRec.Fields(RECORDS_fld_IS_CONSTITUENT) = "True"
    oRec.Fields(RECORDS_fld_LAST_NAME) = "Bakker"
    oRec.Save

    'Always clean up. Objects with an Init() method typically
     '    have a CloseDown() method.
    oRec.CloseDown
    set oRec = Nothing
```

Since the CRecord object (remember, CRecord is the data object that represents a constituent in ***The Raiser's Edge***) has only one required field, we are able to initialize a new object, set the contents of the **Last name** field, and issue a save. All top level objects are added in this exact same manner. Earlier in this guide, we discussed child objects and the object hierarchy. You access child objects only via a parent (top level) object. Therefore, child objects are added in a slightly different fashion. For more information about adding and deleting child objects in detail, see "Programming Child Objects and Collections" on page 21.

Deleting a Record Using a Data Object

You learned how to load a data object from the database (see "Loading a Data Object" on page 15 for more information). Deleting a record requires just one more line of code. This code sample builds on an earlier sample where we learned how to load a data object.

```
Dim oConstit as CRecord
    Set oConstit = New CRecord

    oConstit.Init oSessionContext

    'Load the record via the Social Security Number
     '  Note: we left out some error trapping here to keep the sample clear
     '       (for example if this record didn't exist)
    oConstit.LoadByField uf_Record_SOCIAL_SECURITY_NO, "025-64-6382"

     'Delete the Record using the Data Object's .Delete method
    oConstit.Delete

    oConstit.CloseDown
    set oConstit = Nothing
```

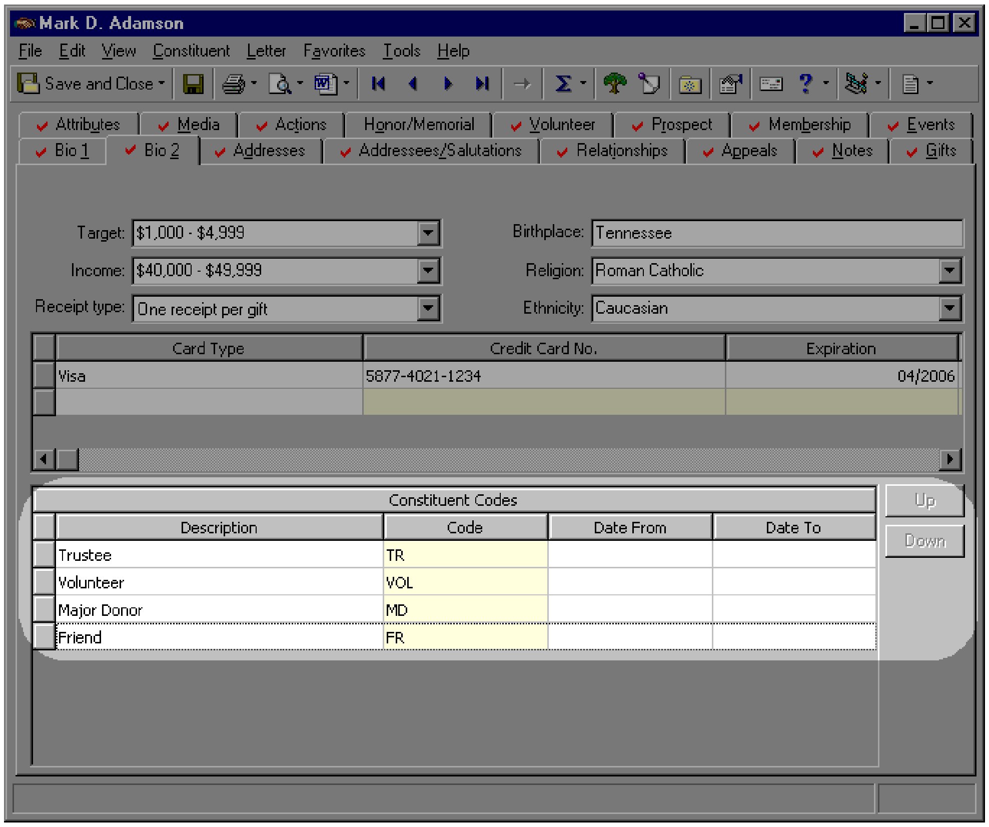
Programming Child Objects and Collections

In this section we examine the details of working with child objects. We also discuss the various types of object collections exposed in ***The Raiser's Edge*** model.

What is a Child Object?

A child object cannot exist without a top-level object. To repeat an earlier example, if you need to add constituent codes to a record in ***The Raiser's Edge***, you must first load a constituent record. When you need to add constituent codes programmatically, you must also first load and initialize the parent record. This is the best way to conceptualize child objects.

In the following graphic, we see ***The Raiser's Edge*** constituent form. The form encapsulates all the child objects of a constituent. Note that the constituent codes are all children of this record and are available only through the constituent's CRecord object.


Mark D. Adamson
File Edit View Constituent Letter Favorites Tools Help
Save and Close
Attributes Media Actions Honor/Memorial Volunteer Prospect Membership Events
Bio 1 Bio 2 Addresses Addressees/Salutations Relationships Appeals Notes Gifts
Up
Down

Target:	$1,000 - $4,999
Income:	$40,000 - $49,999
Receipt type:	One receipt per gift
Birthplace:	Tennessee
Religion:	Roman Catholic
Ethnicity:	Caucasian

Card Type	Credit Card No.	Expiration
Visa	5877-4021-1234	04/2006

Constituent Codes

Description	Code	Date From	Date To
Trustee	TR		
Volunteer	VOL		
Major Donor	MD		
Friend	FR		

Child objects cannot be created, loaded, saved, initialized, or deleted. All these actions are accomplished via methods exposed by the child object's parent object in the hierarchy.

Child Collection Types

Now that you understand the concept of child objects, you can master the details. Not all child objects are the same. The various types of child objects and collections and the mechanics of programming objects and collections differ.

- Collection Type 1 - "The Standard Child Collection" on page 22
- Collection Type 2 - "The Child Top Collection" on page 26
- Collection Type 3 - "The Child View Collection" on page 28
- Collection Type 4 - "The Top View Collection" on page 28

The Standard Child Collection

The most common use of child objects in ***The Raiser's Edge*** object model is via child collections. A child collection, which is collection of child objects, cannot exist without a top-level object. You can add and remove child objects from the collection, but you cannot save child objects without calling the parent's save method.

Why does this seemingly artificial constraint exist? Child objects depend on the parent's save method because the parent may have to enforce rules governing membership in the collection. When the parent is saved, all the child objects in the collection are saved if they are dirty (meaning either their data has been changed since they were loaded, or they have been newly added), and all the objects that have been removed from the collection are deleted from the database.

Common Structure Shared By All Child Collections

If you are familiar with programming using *Visual Basic* collections, then the methods and means for programming ***Raiser's Edge*** child collections should seem quite natural. The following table lists the common methods and properties available on every child collection in ***The Raiser's Edge*** object model.

Method	Description
Item	Returns a child object given an index.
Add	Creates a new child object, stores its membership in the collection, and returns a reference to it.
Remove	Removes a child object from the collection. Once a child object is removed from a collection it cannot be used.
IBBMoveServer	Optional: This establishes how the "VCR" buttons on the form function. This is covered in detail in Programming Reference.
Count	Provides a count of the number of child objects in the collection.

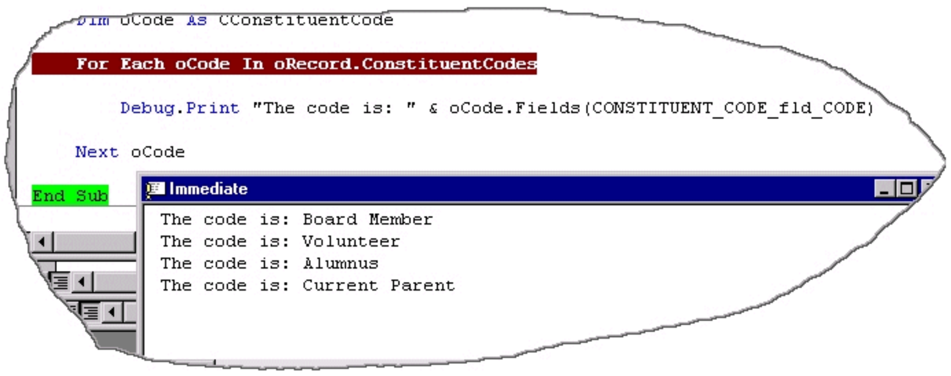
Following standard collection object model design practices, ***The Raiser's Edge*** always has two closely related classes that handle exposing collections: the parent, which is always named in the plural form (for example, ConstituentCodes), and the child, which is always named in the singular form (for example, CConstituentCode).

Navigating Child Object Collections

The easiest, most efficient manner for navigating through (also referred to as *iterating*) child collections is through the *Visual Basic* "For Each" syntax. All collections support this format. Review the sample code below.

```
'Code to initialize and load a CRecord object (oRecord) omitted for brevity.

    Dim oCode as CConstituentCode

    For Each oCode In oRecord.ConstituentCodes

        Debug.Print "The code is: " & oCode.Fields(CONSTITUENT_CODE_fld_CODE)

    Next oCode
```

The code sample loops through each constituent code found linked to the constituent currently being referenced by the oRecord variable. When the last code is accessed, the loop automatically terminates. Here is a sample of the output from the code above.



```
Dim oCode As CConstituentCode

For Each oCode In oRecord.ConstituentCodes

    Debug.Print "The code is: " & oCode.Fields(CONSTITUENT_CODE_fld_CODE)

Next oCode

End Sub
```

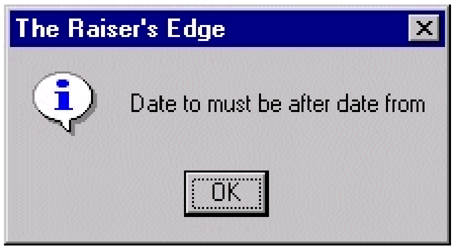
```
The code is: Board Member
The code is: Volunteer
The code is: Alumnus
The code is: Current Parent
```

Updating Child Collection Elements

The previous graphic illustrates how easy it is to access the members of a child collection. To modify the child objects, we need only to add a line of code that updates the child data via its Fields property.

```
'Code to initialize and load a CRecord object (oRecord) omitted for brevity.

    Dim oCode as CConstituentCode

    For Each oCode In oRecord.ConstituentCodes

        Debug.Print "The code is: " & oCode.Fields(CONSTITUENT_CODE_fld_CODE)

        'Modify each constituent code, setting it's date
         '    from to December 13th, 1967
        oCode.Fields(CONSTITUENT_CODE_fld_DATE_FROM) = "12/13/1967"

    Next oCode

    'Important!  None of the changes are saved until
     '    the next line of code executes!!
    oRecord.Save
```

It is very important to note that the constituent code changes were not immediately committed to the database. Remember, child objects do not have a save method; the top-level parent (in this case the oRecord object variable) is responsible for the save. When we issue oRecord.Save, all the changes made are validated against any system and end-user Business Rules. If the rules "pass" all the rule checks, the records are committed to the database. If a rule fails to be validated, ***The Raiser's Edge*** raises a trappable runtime error. For example, one of the rules that apply to constituent codes is the **Date from** field must come before (chronologically) the **Date to** field. So, in the example above, if we had a constituent code child element that already has a date to of "12/12/1967", ***The Raiser's Edge*** would yield the following error.



Date to must be after date from

As we mentioned earlier in this guide, this internal checking is paramount to preserving database integrity. You simply cannot "corrupt" your ***Raiser's Edge*** data from *Visual Basic* code. The same rules that apply to an end-user apply to your data objects.

Accessing Specific Child Elements

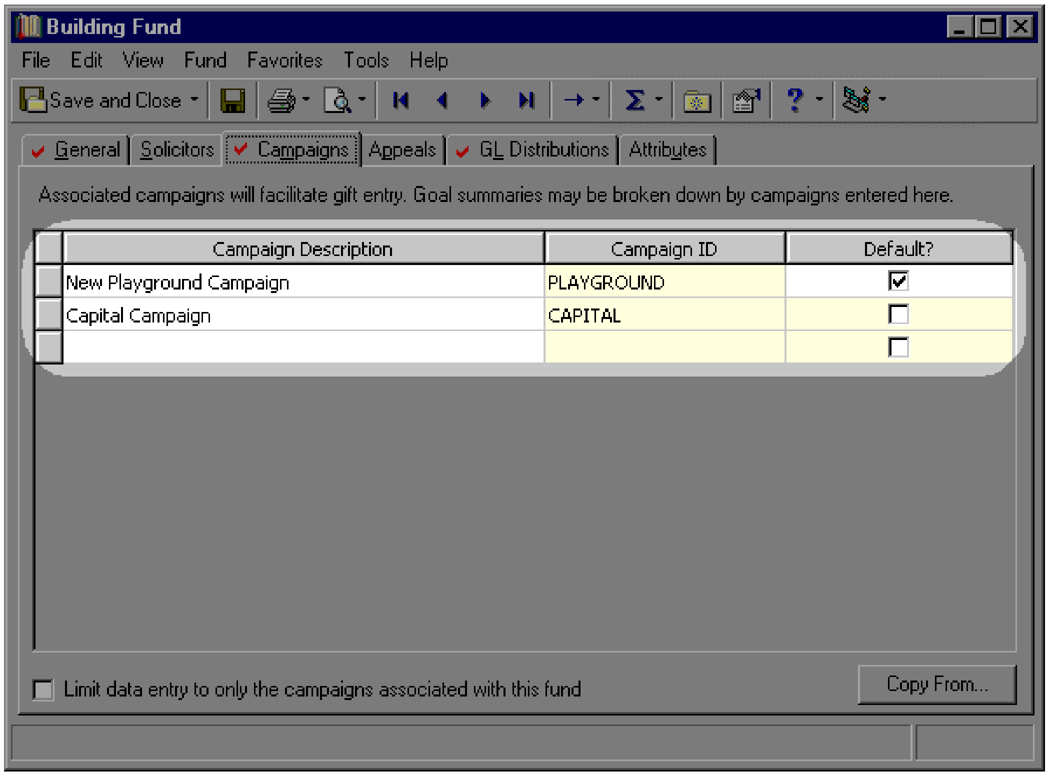
Like any *Visual Basic* collection, ***Raiser's Edge*** child objects can be accessed directly via the Item method. Things become more advance here. Since a ***Raiser's Edge*** child collection is providing high-level access to underlying database records, the developers at Blackbaud needed to "overload" the behavior of the item method, providing multiple ways to use it depending on the context of how it is accessed. For example, if you pass in a string (For example, "66"), the Item method returns the child object whose database ID field is equal to 166. If you pass in a number (For example, 166), then the item method will return the "nth" member of the collection.

```
'Access the 5th element in the collection
    With oRec.ConstituentCodes.Item(5)
        Debug.Print .Fields(CONSTITUENT_CODE_fld_DATE_FROM)
    End With

    'Access an element of the collection that has an
     '    underlying database id (primary key) of 5.
    With oRec.ConstituentCodes.Item("5")
         Debug.Print .Fields(CONSTITUENT_CODE_fld_DATE_FROM)
    End With
```

To ensure consistent access to collections across the object model, we provide these two different methods. The most common use of the Item method of a child collection is to pass it a numeric parameter, accessing the nth item. This becomes more evident when we discuss sorting later in this section. When we work with top-level collections, the value of accessing collection elements via the database ID becomes more clear.

The Child Top Collection

The child top collection is slightly different from a standard child collection. The first major difference lies in the fact that the child top collection is a collection of top-level objects accessible via another top-level object. The collections we have looked at so far are children of a specific top-level object; the child elements cannot exist without the parent. What if top-level objects are related in ***The Raiser's Edge***? This is when child top collections come into use. For example, a campaign object in ***The Raiser's Edge*** is a top-level object, but campaigns can be linked to multiple funds, which are also top-level objects.

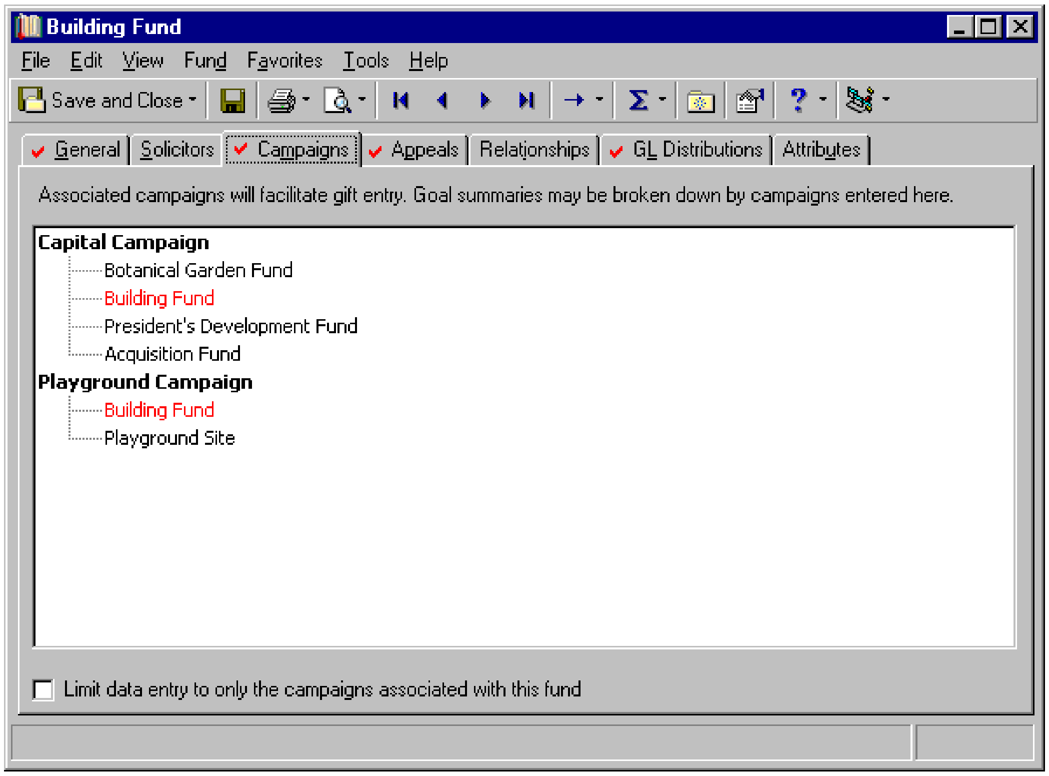

Building Fund
File Edit View Fund Favorites Tools Help
Save and Close
General Solicitors Campaigns Appeals GL Distributions Attributes
Copy From...

Associated campaigns will facilitate gift entry. Goal summaries may be broken down by campaigns entered here.

Campaign Description	Campaign ID	Default?
New Playground Campaign	PLAYGROUND	☑
Capital Campaign	CAPITAL	☐
		☐

☐ Limit data entry to only the campaigns associated with this fund

With ***The Raiser's Edge Enterprise***, the Campaigns tab is not a child top collection. It is view only.


Building Fund
File Edit View Fund Favorites Tools Help
Save and Close
General Solicitors Campaigns Appeals Relationships GL Distributions Attributes

Associated campaigns will facilitate gift entry. Goal summaries may be broken down by campaigns entered here.

Capital Campaign
- Botanical Garden Fund
- Building Fund
- President's Development Fund
- Acquisition Fund

Playground Campaign
- Building Fund
- Playground Site

☐ Limit data entry to only the campaigns associated with this fund

This is the standard fund record form presented by ***The Raiser's Edge***. In this graphic, the end-user has navigated to the Campaigns tab of a fund record. The campaigns linked to the Building Fund are highlighted. To link another campaign to the fund, the end-user would simply click the binoculars in the grid and an Open search screen would appear with a list of campaigns that exist in the database. From here, the end-user could select an additional campaign to link to the fund. The important thing to remember is the user is not entering any "new" data. She is simply linking this fund to an existing campaign. The same applies to programmatic manipulation of a child top collection. The child top collection has an "add" method just like any collection. However, the "add" method of a child top collection accepts one argument, which is a reference to an existing top-level object. The code sample below explains this.

```
'Load fund record 1
    Dim oFund as CFund
    Set oFund = New CFund

    oFund.Init REApplication.SessionContext
    oFund.Load 12

    'Get the first campaign in the system
    '    by navigating to the first element in top level campaigns collection
    Dim oCamps As CCampaigns
    Set oCamps = New CCampaigns

    oCamps.Init REApplication.SessionContext

    Dim oCamp As CCampaign
    Set oCamp = oCamps.Item(1)

    'Add the campaign to this fund's Campaigns collection
    oFund.Campaigns.Add oCamp
    '...Cleanup code omitted
```

The Child View Collection

Child view collections are collections that allow you to navigate to a subset of a particular true child collection. You cannot add to or remove from these collections using standard collection methods because they are just views of another collection. Their membership is determined by other factors (many times, by very specific methods on the parent object). A good example of a child view collection is the InstallmentPayments collection exposed by the gift object. Due to the many issues involved in paying off pledge installments, it is not practical for the mechanics of paying a pledge to be handled via a simple add method. A specific object is used to handle this process, the PledgePayer (this is discussed later in the guide). Child view collections have the following methods.

Method	Description
Item	Returns a child object given an index.
Count	Provides a count of the number of child objects in the collection.

The Top View Collection

The top view collection is a collection in which you to enumerate a top-level object. These objects are useful if you want to process all the instances of a given object in the system. Top view collections have no add or remove methods since adding and removing top-level data objects is achieved through methods on the top-level objects themselves.

One powerful feature of top view collections is the ability to apply a filter to the collection when it is initialized so that only a specific subset of objects are included. For example, you may want to only include active campaigns when using the CCampaigns collection. In this case, you pass the correct filter constant (tvf_Campaign_ActiveOnly) and as the collection is initialized, it contains only campaigns that have been marked as active. This additional parameter is optional.

```
'Define a variable to navigate the top view collection.
    Dim oAllCamps as CCampaigns
    Dim oCamp as CCampaign

    Set oAllCamps = new Ccampaigns
    oAllCamps.Init REApplication.SessionContext, tvf_Campaign_ActiveOnly

    For Each oCamp in oAllCamps
        Debug.Print oCamp.Fields(Campaign_fld_Description)
    Next oCamp
```

Each top-level object has a corresponding top view collection. Remember, a distinguishing characteristic of a collection is that the object's name takes the plural form. The table below lists all the top-level objects and their corresponding collections. The **Record Type** column refers to the record as it is represented in ***The Raiser's Edge***. The **Data Object** column lists the corresponding data object that is used to manipulate the record type programmatically. The **Collection Object** column lists the top view collection object that can be used to navigate the record's top view collection.

Record Type	Description	Collection Object
Constituent	CRecord	CRecords
Gift	CGift	CGifts
Action	CAction	CActions
Fund	CFund	CFunds
Campaign	CCampaign	CCampaigns
Appeal	CAppeal	CAppeals
Membership	CMembership	CMemberships
Job	CJob	CJobs
Special Event	CSpecialEvent	CSpecialEvents

Adding and Deleting Child Objects

Now that we have discussed child objects and the different types of child collections, we can look at how to add or delete a child object. Although there are many different types of child objects, the process to add or remove a child object is the same. It is important to remember that the changes are not actually made to the database until the parent record's save method is called.

Please remember....

It is important to remember that changes are not actually made to the database until the parent record's Save method is called.

Adding a Child Object

To create a new top-level object, you have to use the VB construct of (for example, Set oRecord = New CRecord). However, this does not work for child objects because child objects cannot be created without a top-level object and are always contained within a child collection. Therefore, the first step is to load the top-level object you want to add to. Next, use the collection's Add method to return a new child object to use. At this point, the object is a member of the collection but is not added to the database until the save method is called. The following code sample explains how this is done.

```
'This initializes and loads the parent record
Dim oRecord As CRecord
Set oRecord = New CRecord

oRecord.Init REApplication.SessionContext

oRecord.LoadByField uf_Record_CONSTITUENT_ID, 75

'Create a CConstituentCode child object
Dim oConsCode As CConstituentCode

'The Add method returns a reference to a new CConstituent Code
' object in the CConstituentCodes collection
Set oConsCode = oRecord.ConstituentCodes.Add

'Set the values for the different fields

oConsCode.Fields(CONSTITUENT_CODE_fld_CODE) = "Current Parent"
oConsCode.Fields(CONSTITUENT_CODE_fld_DATE_FROM) = "1/1/1998"

'This step saves the new constituent code information to the database
oRecord.Save

'Clean up!
oRecord.CloseDown
Set oRecord = Nothing
```

The important point to remember is that the Add method of the collection is the only way to create a new child object. All child objects are added using the same process.

Deleting a Child Object

Deleting a child object is very similar to adding a child object. First, you must load the parent object and then call the collection's remove method. This removes the child object from the collection you must call the parent's save method before the object is actually removed. Similar to the Item method discussed earlier, the Remove method is overloaded, providing two different ways to specify the child object to remove. The Remove method accepts either the actual object or the object's Item number as parameters. Review an example of both.

```
'This removes oConstituentCode from the collection
    oRec.ConstituentCodes.Remove oConstituentCode

     'This removes the 2nd element from the collection
    oRec.ConstituentCodes.Remove 2

     'The object is not actually removed from the database until this step
    oRec.Save
```

Either method accomplishes the same task. The situation determines the best method to use. When you remove a child object, no warning message appears, so you should add a warning that the end-user is about to delete information.

Sorting Collections

After you know how to access and move through collections, you may want to arrange the objects in a different order from the way they normally appear in the collection. Not all collections can be sorted in this manner, but many of the more commonly used collections do support sorting.

When sorting collections, you must keep a couple of very important things in mind.

- First, when using the Item method, remember that it returns the nth member based on the current sort.
- Second, when using top view collections, it is possible to filter out top-level objects using a query.

If you filter the collection using a query, the query order is retained regardless of following settings.

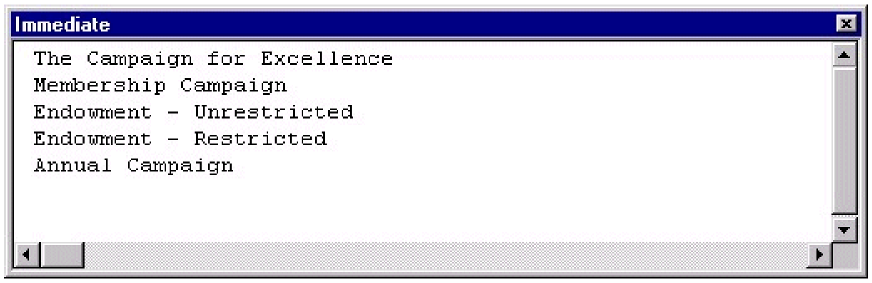
SortField

You can use the SortField property to specify any data field available in the member object to be the field used to sort the entire collection. With the use of IntelliSense and Enums, it is very easy to choose the field you would like to sort by.

SortOrder

The SortOrder property allows you to sort in either ascending or descending order. If no SortOrder is specified, then the default order is ascending.

```
'Initialize collection
    Dim oFunds As CFunds
    Set oFunds = New CFunds
    oFunds.Init REApplication.SessionContext

     'Set the Field and Order for the sort
    oFunds.SortOrder = Descending
    oFunds.SortField = FUND_fld_DESCRIPTION

     'Loop through the collection
    For Each oFund In oFunds
        Debug.Print oFund.Fields(FUND_fld_DESCRIPTION)
    Next oFund
```



```
The Campaign for Excellence
Membership Campaign
Endowment - Unrestricted
Endowment - Restricted
Annual Campaign
```

Filtering Data Object Collections

Collections contain many methods and properties that make it easy to move through them to gather the information you need. If you do not need to see all the child objects in a collection, you can use a query to filter the child objects in your collection. In top-level collections, you can filter the child objects in the collection based on a query.

To filter a top-level collection, use the query ID of the query you want to filter. To get this query ID, you can use the LoadByField method covered in “Loading a Data Object” on page 15. Otherwise, you can iterate through the QueryObjects collection to find the query. The query type must match the record type of the collection. For example, if you are using a CRecords collection, the query you use must be a Constituent query. If you specify a query of the wrong type, a trappable error message appears.

Once you know the query ID, set the property FilterQueryID equal to this query ID. The collection returns only child objects contained in that query. Note the child objects are sorted into the collection in the same order as in the query.

```
Dim oQuery As CQueryObject
     Set oQuery = New CQueryObject

    oQuery.Init REApplication.SessionContext

     'This loads the query that is named Major Donors.
    oQuery.LoadByField uf_QUERY_NAME, "Major Donors"

    Dim oRecords As CRecords
    Set oRecords = New CRecords

    oRecords.Init REApplication.SessionContext

     'This tells the collection which query (Major Donors) to filter with.
    oRecords.FilterQueryID = oQuery.Fields(QUERIES2_fld_QUERIESID)

    'From here on, we can use the oRecords collection and it will only
     '    contain cRecord objects that are in the Major Donor query.
```

Error Handling

Before you resolve errors generated during program processing, it is important to understand the possible ways ***The Raiser's Edge*** objects can "communicate" with your programming. As you program, many times a ***The Raiser's Edge*** object needs to return information to your programs. For example, using a query to filter the objects in our collection. If you tried to use a campaign query to filter a fund collection, this would not work. The object needs some way to communicate this back to the program, so that you can resolve this problem. You can use two methods to accomplish this:

- "Return Code Based" on page 33
- "Error Code Based" on page 33

Return Code Based

If you use return values, object methods are set up to return an error code if they are unsuccessful. Advantages to this include enabling you (in fact, almost forcing you) to handle every possible error as it happens. The downside is that it can be cumbersome to explicitly check for every possible error throughout your code.

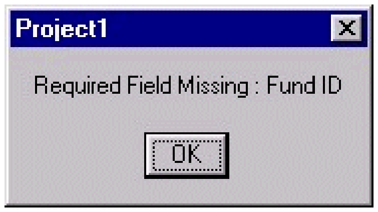
Error Code Based

The alternative is to use *VB's* built-in capability to raise errors, which is what ***Raiser's Edge*** objects do. If proper error handling is not in place, these errors can cause our program to abort. Fortunately, handling errors in *VB* is very simple and offers many flexible ways to resolve errors. Depending on how you structure error handlers, you can handle each error in the subroutine in which it occurs, allow it to cascade back to a central error handler for the entire program, or use a variation of the two.

When an error is raised, we can access information about the error by using the Err object provided by *VB*. Err.Description is a helpful property that tells you the reason for the error, such as failing to specify all required fields when adding a ***Raiser's Edge*** new fund through code.

```
Dim oFund As CFund
    Set oFund = New CFund

    oFund.Init REApplication.SessionContext
    oFund.Fields(FUND_fld_DESCRIPTION) = "The Sullivan Scholarship Fund"

    On Error GoTo ErrorHandler

    oFund.Save

     'This turns the Error Handler off.
    On Error GoTo 0

     'Clean up!
    oFund.CloseDown
    Set oFund = Nothing

    'Always place an Exit Sub before the Error Handler
     '    to prevent entering the Error Handler unintentionally.
Exit Sub

ErrorHandler:
    MsgBox Err.Description, vbOKOnly
     'This returns processing back to the line after where the error occurred
    Resume Next
End Sub
```

If we ran this code, the CFund object would raise an error and we would get a message box similar to the one below.



Required Field Missing : Fund ID

User Interface (UI) Objects

As you learned previously, it is very easy to access, change, and save data using ***Raiser's Edge*** objects. To get the information you need from the end-user, you must have some sort of user interface (UI). In many instances, you need to design a custom form in order to accomplish your design goals. In some cases, you may want to use a form that already exists in ***The Raiser's Edge***. Using an existing ***Raiser's Edge*** form in your project is a simple way to save programming time and makes your program easier to use because the forms are familiar.

Some ActiveX controls in ***The Raiser's Edge*** are very convenient for displaying certain types of information. For example, all of the different types of attributes are displayed in a specific control, called the **Attributes** grid. In your program, if you need to display a list of attributes, you can use the same control ***The Raiser's Edge*** developers use to display attributes.

Data Entry Forms

Because your end-users are already familiar with data entry forms, you can drastically simplify your programming by using the same forms as those used by ***The Raiser's Edge*** developers. All the top-level objects have UI forms. When a form appears, it is fully functional and contains all toolbars and menu bars so end-users can perform the same operations they normally do within the program.

Showing a Standard Form

You can use many different UI forms in the program. Although different forms may have some different methods and properties depending on their use, forms do have some things in common.

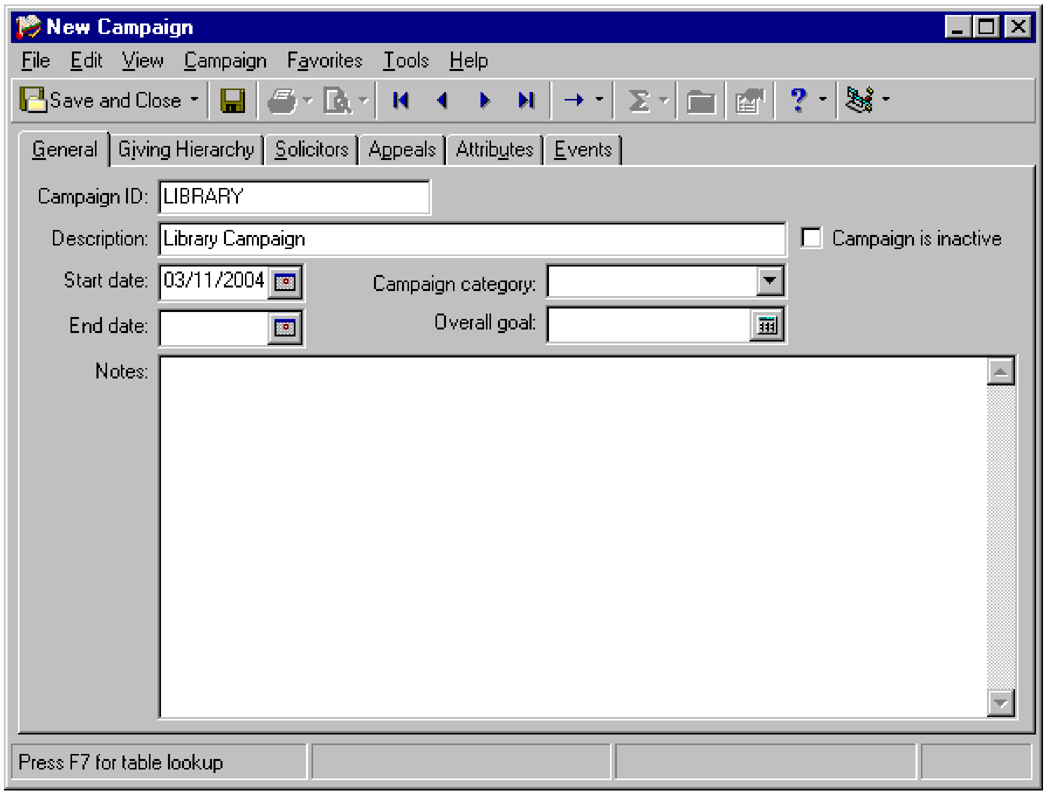
- They always have an Init method which accepts a SessionContext and a CloseDown method.
- They have a property that accepts a data object that "matches" the form. For example, the CCampaignForm.CampaignObject needs a CCampaign object. This is important because the UI form needs to have a reference to a data object so that it can make changes or create a new record based on the actions of the user. If the data object passed is a new data object, then a new record is created if the user chooses to save while using the form. If an existing data object is passed, then changes the end-user makes are saved to the existing record if the end-user chooses to save while using the form. By using the Fields property of the data object, you can fill in some of the fields on the form before it is displayed to the end-user.
- They always have the ShowForm method. This is what actually displays the form. ShowForm accepts the following four optional parameters.

Parameter	Variable Type	Description
bModal	Boolean	Determines whether the form displays modally, defaults to False.
oFormToCenterOn	Object	The UI Form displays itself centered over the form specified here.
bDoNotCloseDataObject	Boolean	If set to True, the data object passed to the form is still initialized and able to used after the user has closed the form.
oMoveServer	IBBMoveServer	This establishes how the "VCR" buttons on the form function. This is covered in detail in Programming Reference.

Code Sample

```
Dim oCampaign as CCampaign
    Set oCampaign = New CCampaign
    oCampaign.Init REApplication.SessionContext

    'If we wanted to use show an existing Campaign, we would load it here.
     '    Or we could set some of the .Fields before we display the Campaign.
    oCampaign.Fields(Campaign_fld_CampaignID) = "LIBRARY"
    oCampaign.Fields(campaign_fld_DESCRIPTION) = "Library Campaign"
    oCampaign.Fields(campaign_fld_START_DATE) = "11/6/98"

    Dim oForm as CCampaignForm
    Set oForm = New CCampaignForm
    oForm.Init REApplication.SessionContext

     'This must be done first or an error will be raised.
    Set oForm.CampaignObject = oCampaign

     'This will display the form modally, centered over frm_Main.
    oForm.ShowForm True, frm_Main

     'Clean up!
    oForm.CloseDown
    Set oForm = Nothing

    oCampaign.CloseDown
    Set oCampaign = Nothing
```

This code displays a fully functional Campaign so the end-user can do anything he normally does on a Campaign form in ***The Raiser's Edge***.


New Campaign
File Edit View Campaign Favorites Tools Help
Save and Close
General Giving Hierarchy Solicitors Appeals Attributes Events

Campaign ID: LIBRARY

Description: Library Campaign

Campaign is inactive

Start date: 03/11/2004

Campaign category:

End date:

Overall goal:

Notes:

Press F7 for table lookup

Raiser's Edge ActiveX Controls

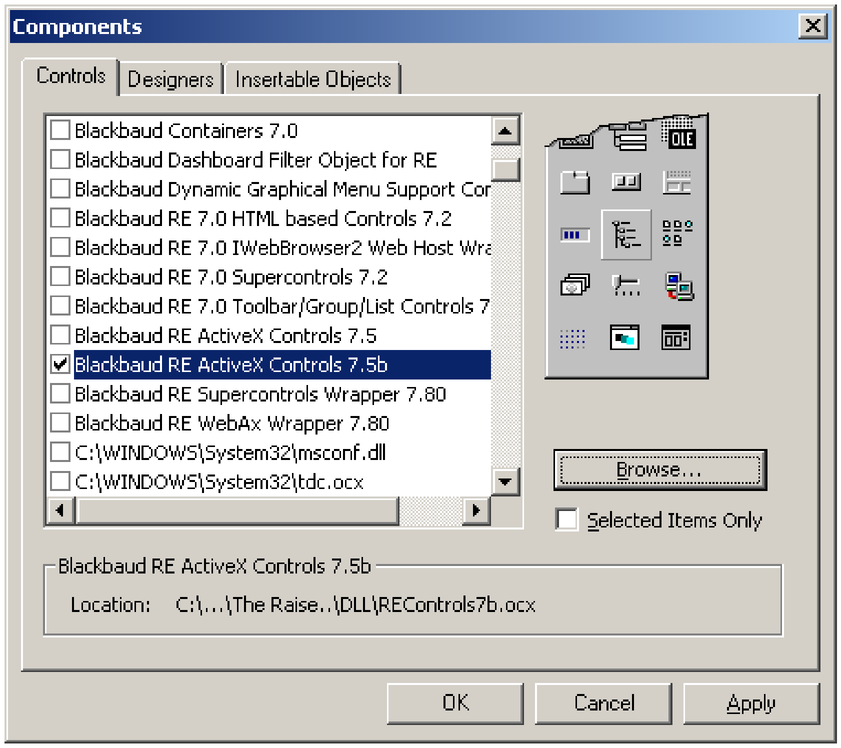
In the previous section, you learned how to use existing ***Raiser's Edge*** forms in your programs to simplify programming and increase usability. Now, you can also learn how to use a few specific controls to simplify your code. When developing ***The Raiser's Edge***, the development team decided to create their own ActiveX controls for displaying certain lists of information, such as attributes. This way, we can design a grid that accomplishes all the design goals for displaying that particular type of information.

Three ActiveX controls can be used to display common ***Raiser's Edge*** collections in a grid.

- The Selection List is used in many different areas. For example, it is used to display a list of gifts, actions, or notepads.
- The **Attributes** grid is used to display the different types of attributes used in ***The Raiser's Edge***.
- The **Phones/Email/Links** grid is used to display phone numbers, email addresses, and links to Web sites.

Due to the specific types of information each of these grids displays, ***The Raiser's Edge*** developers were able to build specific functionality into the control, making it easy to incorporate those same functions into your programs.

To use these controls in *Visual Basic 6.0*, first add the components to your VB project. To do this, select **Project**, **Components** from the menu bar. When the Components form displays, mark **Blackbaud RE ActiveX Controls 7.5b**.



- [ ] Blackbaud Containers 7.0
- [ ] Blackbaud Dashboard Filter Object for RE
- [ ] Blackbaud Dynamic Graphical Menu Support Cor
- [ ] Blackbaud RE 7.0 HTML based Controls 7.2
- [ ] Blackbaud RE 7.0 IWebBrowser2 Web Host Wra
- [ ] Blackbaud RE 7.0 Supercontrols 7.2
- [ ] Blackbaud RE 7.0 Toolbar/Group/List Controls 7
- [ ] Blackbaud RE ActiveX Controls 7.5
- [x] Blackbaud RE ActiveX Controls 7.5b
- [ ] Blackbaud RE Supercontrols Wrapper 7.80
- [ ] Blackbaud RE WebAx Wrapper 7.80
- [ ] C:\WINDOWS\System32\msconf.dll
- [ ] C:\WINDOWS\System32\tdc.ocx

- [ ] Selected Items Only

Blackbaud RE ActiveX Controls 7.5b

Location: C:\...\The Raise..\DLL\REControls7b.ocx

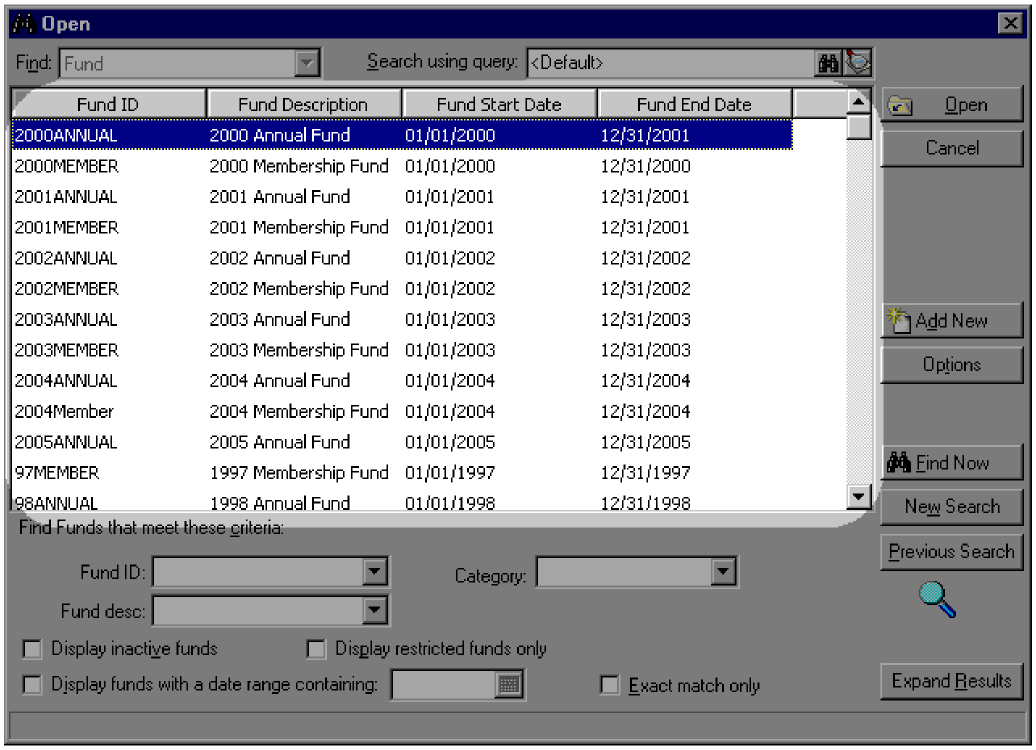
Note...

If you are using *VBA*, the control references are REControls7b.REAttributeGrid, REControls7b.REDataGrid, and REControls7b.REPhoneGrid. For more information about *VBA*, see the VBA chapter in this guide.

Once you have added the Controls to your project, the next step is to place the control you want on the form.

Data Grid

The data grid is used in many places to display information and enable the user to select a record from the list in ***The Raiser's Edge***. For example, it is used to display lists of constituents, funds, and gifts. After you have drawn the grid onto your form, you may want to set the property PreviewDataType. This property, which is only available at design-time, allows you to tell the control the type of data objects you want to display. It then shows all the columns that are standard for that type of record. You can then size the grid appropriately. You do not have to set this property during design-time. The grid displays the columns based on the type of collection you use regardless. However, it is an easy way to determine the best size for the grid.



Find: Fund
Search using query: <Default>

Fund ID	Fund Description	Fund Start Date	Fund End Date
2000ANNUAL	2000 Annual Fund	01/01/2000	12/31/2001
2000MEMBER	2000 Membership Fund	01/01/2000	12/31/2000
2001ANNUAL	2001 Annual Fund	01/01/2001	12/31/2001
2001MEMBER	2001 Membership Fund	01/01/2001	12/31/2001
2002ANNUAL	2002 Annual Fund	01/01/2002	12/31/2002
2002MEMBER	2002 Membership Fund	01/01/2002	12/31/2002
2003ANNUAL	2003 Annual Fund	01/01/2003	12/31/2003
2003MEMBER	2003 Membership Fund	01/01/2003	12/31/2003
2004ANNUAL	2004 Annual Fund	01/01/2004	12/31/2004
2004Member	2004 Membership Fund	01/01/2004	12/31/2004
2005ANNUAL	2005 Annual Fund	01/01/2005	12/31/2005
97MEMBER	1997 Membership Fund	01/01/1997	12/31/1997
98ANNUAL	1998 Annual Fund	01/01/1998	12/31/1998

Find Funds that meet these criteria:

Fund ID:
Category:
Fund desc:

☐ Display inactive funds
☐ Display restricted funds only
☐ Display funds with a date range containing:
☐ Exact match only

Now, let’s review the grid during run-time. Before we can use the grid, we must call the Init method. Next, we set the DataCollection property by passing a collection of the data objects we want to display. The last step to displaying the grid is to call the Refresh method. Any time your data collection changes, you should call the Refresh method. This updates the grid using the current members of the collection. The following code sample shows you how to use the grid.

```
Option Explicit
Private moRecord as CRecord

Private Sub Form_Load()

    Set moRecord = New CRecord
    moRecord.Init ReApplication.SessionContext
    moRecord.Load 166

    REDataGrid1.Init REApplication.SessionContext
    Set REDataGrid1.DataCollection = oRecord.Gifts
    REDataGrid1.Refresh

End Sub

Private Sub Form_Unload(Cancel As Integer)

    REDataGrid1.CloseDown

End Sub
```

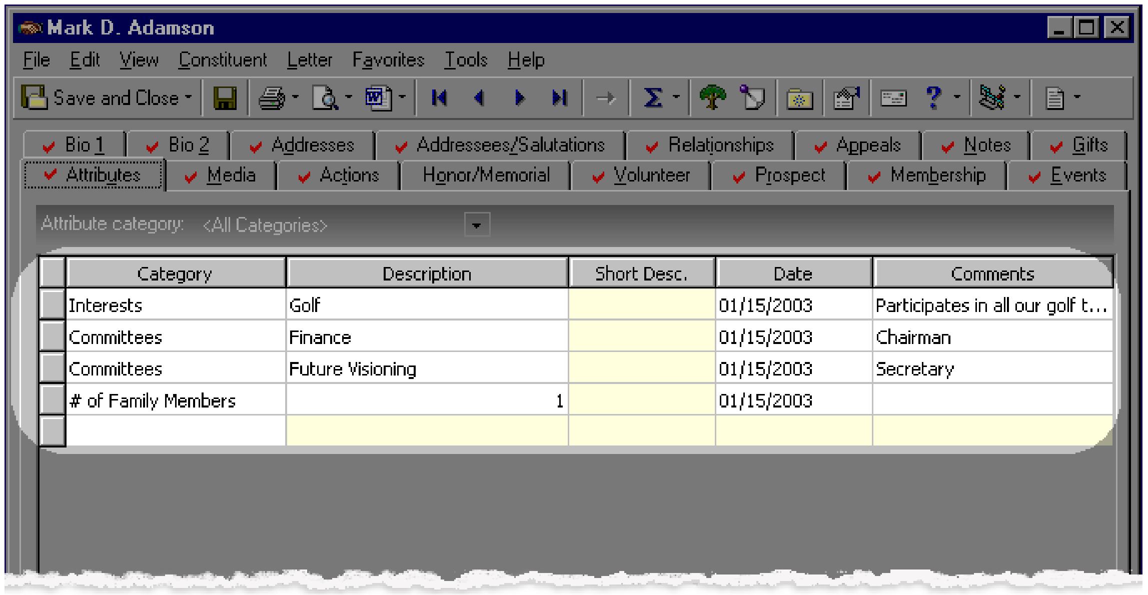
There are methods for the grid that offer access to the same features present in ***The Raiser’s Edge***. Calling the CustomizeCollection method brings up a form the end-user can use to select columns and sort the order in which they appear. Using the ShowLegend method enables the user to select colors, use Bold and Italic type, and add subscripts for certain data objects.

There are also events associated with the grid that allow your end-users to open records from the data grid. For example, there is a double-click event that passes the ID of the selected record, so you can load and display the record. The following code is an example of this.

```
Private Sub REDataGrid1_DoubleClick(ByVal lSelectedID As Long)

    Dim oGiftForm As CGiftForm
    Set oGiftForm = New CGiftForm
    oGiftForm.Init REApplication.SessionContext

    Dim oGift As CGift
    Set oGift = New CGift
    oGift.Init REApplication.SessionContext

     'This uses the ID that is passed by the event to load the correct gift
    oGift.Load lSelectedID

    Set oGiftForm.GiftObject = oGift
    oGiftForm.ShowForm False, Me, True

     'Clean up!
    oGift.CloseDown
    Set oGift = Nothing

    oGiftForm.CloseDown
    Set oGift = Nothing

End Sub
```

Attributes Grid

Unlike the Selection List, the **Attributes** grid allows you to display attributes and enables your end-user to make changes to those attributes.



Category	Description	Short Desc.	Date	Comments
Interests	Golf		01/15/2003	Participates in all our golf t...
Committees	Finance		01/15/2003	Chairman
Committees	Future Visioning		01/15/2003	Secretary
# of Family Members	1		01/15/2003	

To use the grid, you must first initialize it using the familiar Init method. Next, you need to pass the grid the collection of attributes to be displayed using the AttributeCollection property. Lastly, have the grid refresh itself using the Refresh method. When you finish with the grid (probably when the form unloads), call the CloseDown method, in order to free up the memory associated with the grid. The following code is an example.

```
Option Explicit

Private moRecord as CRecord

Private Sub Form_Load()

    Set moRecord = new CRecord
    moRecord.Init ReApplication.SessionContext
    moRecord.Load 166

    With REAttributeGrid1
        .Init REApplication.SessionContext
        Set .AttributeCollection = oRecord.Attributes
        .Refresh
    End With

End Sub

Private Sub Form_Unload(Cancel As Integer)

   REAttributeGrid1.CloseDown

End Sub
```

This is all you need to display a list of attributes. It is important to remember the attribute collection and the grid do not automatically update when one changes. For example, if attributes are added to the collection, the grid does not automatically display those new attributes. In this case, you need to call the Refresh method again. This method updates the grid with the current information from the attributes collection. If your end-user makes changes to the attributes listed on the grid, these changes do not automatically update in the collection. In order to save those changes, call the SaveGridToCollection method. This updates the collection with the end-user’s changes. To actually save the records to the database, you need to call the Save method for the Parent record of the attributes collection. Use the following code sample to do this.

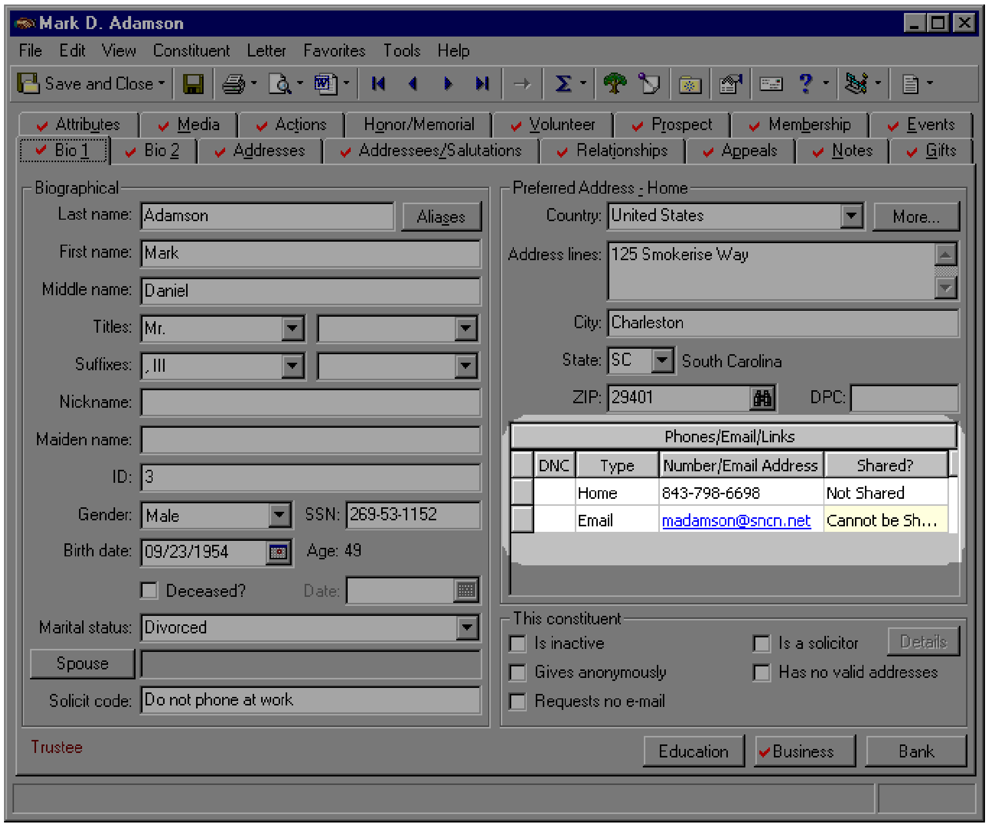
```
'This updates the oRecord.Attributes collection with
    '    the user's changes to the grid
    REAttributeGrid1.SaveGridToCollection

    'The changes are not saved to the database until the
    '    parent record is saved
    On Error GoTo ErrorHandler
        oRecord.Save
    On Error GoTo 0
```

The **Attributes** grid also has a method that handles situations if invalid data is in the collection when it is saved. For example, if an invalid date is entered for an attribute, a trappable error is generated when you save the parent record. By using the HandleError method, you can restore focus to the invalid field. The following code is an example of how to do this.

```
ErrorHandler:

    'This will display a message box, so the user will know what the error is
    MsgBox Err.Description

    'This checks to see if the error is caused by the data in a data object
    If Err.Number = bbErr_DATAOBJECTERROR Then

        'This uses the ErrorObject to determine which part of the record
         '    caused the error
        If TypeOf moREApi.SessionContext.ErrorObject.InvalidObject _
           Is IBBAttribute Then
            'When we pass the Err.Number it will return focus to the
             '    incorrect field on the grid
            REAttributeGrid1.HandleError Err.Number
        End If

    End If
```

Phones/Email/Links Grid

The **Phones/Email/Links** grid has many features in common with the **Attributes** grid. They are both designed to enable the user to input or update information. They also have many of the same properties and methods; therefore, much of the code is familiar. The **Phones/Email/Links** grid automatically formats phone numbers according to the format specified in *Configuration*.


Mark D. Adamson
File Edit View Constituent Letter Favorites Tools Help
Save and Close
Attributes Media Actions Honor/Memorial Volunteer Prospect Membership Events
Bio 1 Bio 2 Addresses Addressees/Salutations Relationships Appeals Notes Gifts
Aliases
More...
Spouse
Details
Education Business Bank

Biographical

Last name:	Adamson
First name:	Mark
Middle name:	Daniel
Titles:	Mr.
Suffixes:	, III
Nickname:	
Maiden name:	
ID:	3
Gender:	Male
SSN:	269-53-1152
Birth date:	09/23/1954
Age:	49
Deceased?	☐
Date:	
Marital status:	Divorced
Solicit code:	Do not phone at work

Preferred Address - Home

Country:	United States
Address lines:	125 Smokerise Way
City:	Charleston
State:	SC South Carolina
ZIP:	29401
DPC:	

Phones/Email/Links

DNC	Type	Number/Email Address	Shared?
	Home	843-798-6698	Not Shared
	Email	madamson@sncn.net	Cannot be Sh...

This constituent

☐ Is inactive ☐ Is a solicitor
☐ Gives anonymously ☐ Has no valid addresses
☐ Requests no e-mail

Trustee

Service Objects

The Raiser's Edge is built on a foundation of programmable objects that provide a high level abstraction over ***The Raiser's Edge*** data model. UI objects enable programmatic access to the system's data entry forms. The wide range of data and UI objects all share a common programming interface. While these objects represent the heart of the system, there is a lot more to ***The Raiser's Edge*** than just data and UI components.

Other objects enable access to discrete functionality within the application. These objects cannot specifically be categorized because they each provide a service via their own unique programming interface. To help organize these entities, Blackbaud's object model refers to them as Service Objects.

In this section we review the service objects exposed by ***The Raiser's Edge*** and examine the mechanics of programming them. It is likely that service objects are called upon frequently as you tackle various development tasks with the system. For example, with the Query service object you can access pre-existing queries, so you can work with the query's result set, opening up a wide range of reporting and data analysis possibilities.

Query Objects

A query object is referred to a group of objects that provide query functionality in ***The Raiser's Edge*** object model. These objects include:

- CQueryObject
- CQueryObjects
- CQuerySet
- CStaticQ

These four objects allow programatic access to existing queries, access to the output of a query, and the ability to create a new static query that can be used elsewhere in ***The Raiser's Edge***. A solid understanding of these objects work goes along way to making your projects faster and more efficient.

Opening a Query

Opening an existing query is quite easy and similar to the data objects that you learned about previously. You access information about a query through the CQueryObject. First, you must initialize the object and then load it. Like data objects, there is a Load method if you know the database ID of the query. There is also a CQueryObjects collection you can loop through to find the correct query. After you load the query, you can access its result set. The following code sample shows how this is done.

```
Dim oQuery as CQueryObject
    Set oQuery = New CQueryObject

    oQuery.Init REApplication.SessionContext

     'Load the query using the Query name
    oQuery.LoadByField uf_QUERY_NAME,"Major Donors Query"

     'Load the query using the Database ID
    oQuery.Load 5
```

Processing a Query Result Set

By processing a query result set, you can move line-by-line through the results of a query. You can access a query result set in two ways.

If we are already using a CQueryObject, we can access its resultset by using the Queryset method:

```
Dim oQueryObject as CQueryObject
    Set oQueryObject = New CQueryObject

    oQueryObject.Init REApplication.SessionContext

     'Load the using the Query name
    oQueryObject.LoadByField uf_QUERY_NAME,"Major Donors Query"

     'This opens the resultset for access
    oQueryObject.QuerySet.OpenQuerySet
```

Or, if you know the query's database ID, you can start with a CQueryset object.

```
Dim oQuerySet as CQuerySet
    Set oQuerySet = New CQuerySet

    oQuerySet.Init REApplication.SessionContext

     'This uses the database ID of the query
    oQuerySet.QueryID = 10

     'This opens the resultset for access
    oQuerySet.OpenQuerySet
```

Both of these examples accomplish the same task. In either case, we have a reference to a query resultset. You can use a few properties that help to actually access the data from the result set:

Property	Returns
FieldCount	The number of fields in the output of the query
FieldName	An array of the field names in the output
FieldType	An array of the field type (for example, Date, Double, Long, Memo, Text)
FieldValue	An array of the actual data for the current row
RowNum	The number of the current row.

Code Sample

```
Debug.Print oQuerySet.FieldName(1) & " " & oQuerySet.FieldName(2)

    Do While Not oQuerySet.EOF
         'This is where you would access the fields
        Debug.Print oQuerySet.FieldValue(1) & " " & oQuerySet.FieldValue(2)
        oQuerySet.MoveNext
    Loop

     'Clean up
    oQuerySet.CloseDown
    Set oQuerySet = Nothing
```

Creating Static Queries

You can create queries through code by using the CStaticQ query service object. Static queries are lists of unique IDs. If you create a static query via code, you cannot open it in *Query* because it has no sort, filter, or output fields. Other queries, and any process that uses a query (such as *Mail* or *Reports*) can use static queries. Static queries are ordered and have no duplicates. To create a static query, you use 3 methods and the Init and CloseDown methods that you use with every object.

1. Use the Create method to create a new query. This method displays the same Create Query form used in ***The Raiser's Edge***. The end-user can specify the name of the query and other information about the query. The Create method returns a False if the user clicks **Cancel**. You should abort your process in this case.The following table shows the parameters for the Create method.

Parameter	Variable Type	Description
SearchType	bbSearch Types	Determines the type of the query. For example, what types of records are included
aFromProcess Name	String	Each query stores from the area of the program it was created. You may put the name of your application here.
FormToCenter On	Object	The Create Query form displays itself centered over the object specified here.
sDescription	String	Optional: Allows you to input a default **Description** for the new query.
lSystemID	Long	Included in parameters, but not applicable.
sDefaultQName	String	Optional: Allows you to input a default **Query name** for the new query.

2. To add the database IDs of the records you want to include in the query, use the AddRecord method and pass the ID as the only parameter. The AddRecord method checks to make sure it is not a duplicate ID and then adds it to the query. This is the only step required to add a record to the query.
3. To finish creating the query and write the information to the database, call the EndCreate method. Until this is called, the IDs are just stored in memory. EndCreate has three parameters:

- FormToCenterOn accepts an object. When EndCreate is called, it normally displays a Writing Static Records form while it is writing the IDs to the database. This parameter specifies the form on which you would like the Writing Static Records form to center itself.
- bCancel is an optional parameter that defaults to False if nothing is passed. If your code allows the end-user to cancel the creation of the query after the Create method is called, it is important to call the EndCreate method and pass True for this parameter. The query is not created, but this frees the memory used to track the IDs for the query.
- bNoUI is an optional parameter. For the program not to display the Writing Static Records, set this to True.

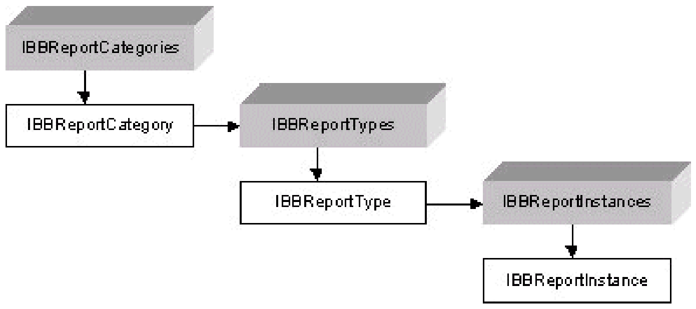
This code sample loops through the records to find the ones you need if want a query of couples in the database who have different last names from each other.

```
Dim oRecord2 As CRecord
Set oRecord2 = New CRecord
Dim oRecord1 As CRecord
Set oRecord1 = New CRecord
Dim oRecords As CRecords
Set oRecords = New CRecords
oRecords.Init oAPI.SessionContext
Dim oStaticQuery As CStaticQ
Set oStaticQuery = New CStaticQ

oRecord2.Init oAPI.SessionContext
oStaticQuery.Init oAPI.SessionContext
'This will prompt the user for a Query Name but
' everything will already be filled in
If oStaticQuery.Create(SEARCH_CONSTITUENT, "Custom App", Nothing, _
    "List of couples who have different last names", , _
    "Spouses W\Different Last Names") Then
    For Each oRecord1 In oRecords
         'This checks first to see if they even have a spouse on their record
        If Val(oRecord1.Fields(RECORDS_fld_SPOUSE_ID)) > 0 Then
            oRecord2.Load oRecord1.Fields(RECORDS_fld_SPOUSE_ID)

            If oRecord1.Fields(RECORDS_fld_SPOUSE_ID) <> "" Then
                 'Compares the constituent's last name with the spouse's last name
                If oRecord1.Fields(RECORDS_fld_LAST_NAME) <> _
                    oRecord2.Fields(RECORDS_fld_LAST_NAME) Then
                      'This adds this ID to our query
                    oStaticQuery.AddRecord oRecord2.Fields(RECORDS_fld_ID)
                End If
            End If
        End If
        oRecord1.CloseDown
    Next oRecord1
    Set oRecord1 = Nothing
    oRecords.CloseDown
    Set oRecords = Nothing
    'Once we have all our records in our query,
     ' we write the data to the database
    oStaticQuery.EndCreate Nothing, False, False
    oStaticQuery.CloseDown
    Set oStaticQuery = Nothing
Else
     'This means the user canceled when entering the query name
    MsgBox "No query created", vbOKOnly
End If
```

Report Objects

Report objects are a group of objects that work together to provide the ability to access ***Raiser's Edge*** *Reports* and *Mail* functionality through code. Since the mail functions also use Crystal Decision's *Crystal Reports*, it makes sense to provide one set of objects that can be used to print reports and process mail functions.

These objects appear in a hierarchy that represent the way they are accessed in ***The Raiser's Edge***. When you access *Reports*, you are presented with a list of categories, such as **Financial Reports**. After you click the category, you are given a list of the different types of reports in that category, such as the **Gift Entry Validation**. When you create a new report, a screen opens containing tabs on which you define parameters for a report. Report objects follow the same hierarchy; however, depending on the needs of your project, you can enter the object model from any object.and create each class independently. You do not directly create report objects as you do the other objects. In order to create a new ReportCategories or ReportCategory object, use the REServices object.



In this hierarchy, IBBReportCategories is a collection of IBBReportCategory objects. These represent the categories of *Reports* or *Mail* options present in ***The Raiser's Edge*** (such as **Financial Reports**, **Action Reports**, or **Letters** in the *Mail* section).

The next level is the IBBReportTypes and the IBBReportType objects; these represent the specific reports (such as the **Gift Entry Validation** and **Action Detail Report**, or **Follow-up Letters** in *Mail*).

The last level of the hierarchy is the IBBReportInstances and IBBReportInstance objects. These correspond to the individual parameter files that you can save for each report. When you are at this level, you can allow your end-user to preview or print the report or even create a new set of parameters for the report type. The next three sections review each of these objects in detail.

Reports Categories Collection

The Raiser's Edge *Reports* and *Mail* breaks down by similar functionality into categories such as **Financial Reports**, **Pledge Reports**, and **Forms** (in *Mail*). Each one of these categories is represented by an IBBReportCategory object. Use the REService object to create both IBBReportCategories and IBBReportCategory objects (for more in formation, see the following "Report Objects Sample" on page 54). After you create an IBBReportCategories collection, use the Init method. This is similar to the Init method used for other objects to initialize them. However, in this case, there are other parameters (see the following table).

Parameter	Variable Type	Description
SessionContext	IBBSessionContext	This is the same SessionContext used to initialize all objects.
lSubCategoryOfCategoryID	Long	Optional: This is not used but is included for future expandability.

Parameter	**Variable Type**	**Description**
CategoryFilter	EReR_ReportCategoryFilters	Optional: This is used to specify *Reports*, *Mail*, or both IBBReportCategory objects in the collection. This defaults to include Report IBBReportCategory objects only.
QueMode	Boolean	Optional: This establishes how errors are handled when using the collection. If it is set to True, a log file is created containing any errors, but the program continues to process. If set to False (which is the default) a trappable error is raised.
ShowMembersBasedOnSecuritySettings	Boolean	Optional: Use this to include IBBReportCategory objects in the collection an end-user has the security to view. If set to False, the collection includes all objects, regardless of the users security. However, at any point the end-user would still be unable to run a report they did not the security to access. This defaults to True.
ShowCannedReportsOnly	Boolean	Optional: There are some IBBReportCategory objects that represent reports not accessed via *Reports*, such as query control reports or **Global Add** reports. If this is set to True (the default), only the IBBReportCategory objects that represent categories found in *Reports* are in the collection.

When you initialize the IBBReportCategories collection, you can use a “For Each” construct to loop through it or use the Item property to access the IBBReportCategory objects in the collection.

If you enter the Report hierarchy directly from an IBBReportCategory object, you need to Init it first. Some Init parameters for the IBBReportCategory are similar to the parameters for the IBBReportCategories, but they work differently. The IBBReportCategory object contains a ReportTypes method. This returns a IBBReportTypes collection of IBBReportType objects. The ShowMembersBasedOnSecuritySettings and ShowCannedReportsOnly parameters filter the IBBReportType objects included in the IBBReportCategory.ReportTypes collection.

Parameter	Variable Type	Description
SessionContext	IBBSessionContext	This is the same SessionContext used to initialize all objects.
CategoryID	EReR_ReportCategories	This is a Enum of all the different categories of reports using in the ***Raiser’s Edge***.

Parameter	Variable Type	Description
QueMode	Boolean	Optional: This establishes how errors are addressed when using the object. If it is set to True, a log file is created containing any errors, but the program continues to process. If set to False (which is the default) a trappable error is raised.
ShowMembersBasedOnSecuritySettings	Boolean	Optional: Use this to include only the IBBReportCategory.ReportTypes collection objects an end-user has the security to view. If set to False, the collection includes all objects, regardless of the user's security. However, at any point the user is unable to run a report he did not have the security to access. This defaults to True.
ShowCannedReportsOnly	Boolean	Optional: Some IBBReportType objects represent reports that are not accessed via *Reports*. If set to True (which is the default), only the IBBReportType objects that represent reports found in *Reports* is included in the IBBReportCategory.ReportTypes collection.

When you initialize the object, you can access the ReportTypes property to move farther down the hierarchy of Report objects. For more information, see the "Report Objects Sample" on page 54 for an example of how to create and use these objects. Refer to Programming Reference to learn more about the other properties and methods of these two objects.

Reports Types Collection

You can access collections and objects that represent the highest level of the Reports hierarchy—the Report Categories. First, use the REServices object to create either an IBBReportTypes or IBBReportType object (see "Report Objects Sample" on page 54). Next, call the Init method. The Init method has some parameters that can filter the IBBReportType objects to be included in the collection (see the following table).

Parameter	Variable Type	Description
SessionContext	IBBSessionContext	This is the same SessionContext used to initialize all objects.
CategoryID	EReR_ReportCategories	This is a Enum of all Report categories so only ReportTypes that are a part of this category are included in the collection.
QueMode	Boolean	Optional: This establishes how errors are addressed when using the object. If set to True, a log file is created containing any errors, but the program continues to process. If set to False (which is the default) a trappable error is raised.

Parameter	Variable Type	Description
ShowMembersBasedOnSecuritySettings	Boolean	Optional: This includes only the IBBReportCategory.ReportTypes collection objects an end-user has the security to view. If this is set to False, the collection includes all objects, regardless of the users security. However, at any point the user is unable to run a report he does not have the security to access. This defaults to True.
ShowCannedReportsOnly	Boolean	Optional: Some IBBReportType objects represent reports that are not accessed via *Reports*. If this is set to True (which is the default), only the IBBReportType objects that represent reports found in *Reports* are included in the IBBReportCategory.ReportTypes collection.

After initializing the IBBReportTypes collection, you can iterate through the collection or select an IBBReportType object by using the Item method.

If you already know the type of report you want to access, enter the report hierarchy at the IBBReportType object. As always, call the Init method and provide a couple of parameters in order to access the correct report (see the following table).

Parameter	Variable Type	Description
SessionContext	IBBSessionContext	This is the same SessionContext used to initialize all objects.
CategoryID	EReR_ReportCategories	This is a Enum of all the Report types so you can specify which report to access.
ShowOnlyMyReports	Boolean	Optional: This establishes how errors are addressed when using the collection. If set to True, a log file is created containing any errors, but the program continues to process. If set to False (which is the default) a trappable error is raised.

When the IBBReportType object is initialized, you can access its read-only properties to get more information about this particular report. It also has a ReportInstances property so you can access the last levels of the Report hierarchy. Here, you can actually process a report.

Report Instances Collection

It is only at the lowest level of the Report object hierarchy that you can process a report. Using the IBBReportInstances and IBBReportInstance objects you can access any parameter files already created and allow end-users to create new ones using the same forms they are used to seeing in ***The Raiser’s Edge***.

The IBBReportInstances object is a collection that represents all the parameter files for a particular type of report. As with the Report objects, it is created using the REServices object (see “Report Objects Sample” on page 54). When you create the object, use the Init method to initialize it. The following table shows the parameters for the Init method.

Parameter	Variable Type	Description
SessionContext	IBBSessionContext	This is the same SessionContext used to initialize all objects.
ReportTypesID	EReR_ReportCategories	This is a Enum of all the Report types so only ReportInstances for the type specified here are included in the collection.
QueMode	Boolean	Optional: This establishes how errors are addressed when using the object. If set to True, a log file is created containing any errors, but the program continues to process. If set to False (which is the default) a trappable error is raised.
ShowOnlyMyReports	Boolean	Optional: If set to True, the collection contains only IBBReportInstances objects that represent parameter files created by the end-user. If set to False (the default), all parameter files available to the end-user are represented in the collection.

After you initialize the collection, you can use any standard process to iterate through the collection.

If you use the IBBReportInstance object to enter the hierarchy, use the Init method. The following table shows the parameters for the Init method.

Parameter	Variable Type	Description
SessionContext	IBBSessionContext	This is the same SessionContext used to initialize all objects.
QueMode	Boolean	Optional: This establishes how errors are addressed when using the object. If set to True, a log file is created containing any errors, but the program continues to process. If set to False (which is the default) a trappable error is raised.

After you initialize an IBBReportInstance, you can either load an existing parameter file or create a new one. To load an existing IBBReportInstance, all you need to know is the ReportParameterID, which is the database ID of the parameter file. After you use the Load method, or if you are creating a new parameter file, call the Process method. The following table explains the parameters for this method.

Parameter	Variable Type	Description
Action	EReR_ProcessOptions	This is an Enum of the process options available.
ShowModal	Boolean	Optional: This determines if the Process form (this varies depending on the action) is displayed modally. This defaults to False.
FormToCenterOn	Object	Optional: This determines which object the Process form displays.

Parameter	Variable Type	Description
RunFromWeb	EReR_WebReportType	Optional: This is used internally for the ***The Raiser's Edge for the Web***. This parameter should be left blank.

The Process method supports a number of actions that are enumerated as EReR_ProcessOptions. These include ReR_ProcessOption_ShowParameterForm which shows the parameter form for the particular report type you are using. If you have not called the Load method, a new parameter form displays that allows the end-user to complete the parameters and save and run the report from the parameter form. If you have called the Load method, the form appears with the parameters already displayed, allowing an end-user to edit the parameters and run the report. If you do not want to display the parameters, you can use the other EReR_ProcessOptions to directly print, print preview, export, send as mail, or view the report layout. The Process method returns a Long integer which is a unique handle to a Crystal Report file. This is used internally by ***The Raiser's Edge for the Web*** and can be disregarded. For more information about other properties and methods available, see Programming Reference.

It is important that when you finish using an IBBReportInstance that you call the CloseDown method. Even though it may return False, indicating that it cannot be closed at this time, it sets an internal flag and cleans everything as soon as the end-user closes the report. For example, after you call the Process method with an action of **Preview**, you can call the CloseDown. When the user closes the preview window or exits the application, the object releases the resources it was using. However, you should make sure you do not need to access any property or method from the object. Once CloseDown is called, the object acts as if it is closed down, even if the preview window or parameter form still displays.

Report Objects Sample

This code sample illustrates how to use the Report objects. In this example, you add all the possible report categories, types, and instances to a treeview.

```
Option Explicit

Private REService As REServices

Private Sub Form_Load()

    Dim oReportCategories As IBBReportCategories
    Dim oReportCategory As IBBReportCategory
    Dim oReportTypes As IBBReportTypes
    Dim oReportType As IBBReportType
    Dim oReportInstances As IBBReportInstances
    Dim oReportInstance As IBBReportInstance


'This is the class that we use to create the Report objects
    Set REService = New REServices
    REService.Init REApplication.SessionContext

    Set oReportCategories = REService.CreateServiceObject(bbsoReportCategories)

    oReportCategories.Init REApplication.SessionContext, , _
                              ReR_ReportCategoryFilter_Reports, False, True, True


For Each oReportCategory In oReportCategories
```

```
TreeView1.Nodes.Add , , oReportCategory.CategoryName, _
                    oReportCategory.CategoryName

         'You could also use oReportCategory.ReportTypes
        Set oReportTypes = REService.CreateServiceObject(bbsoReportTypes)
        oReportTypes.Init REApplication.SessionContext, _
                oReportCategory.CategoryID, False, True, True

        For Each oReportType In oReportTypes
            TreeView1.Nodes.Add oReportCategory.CategoryName, _
                         tvwChild, "Type" & Str$(oReportType.ReportID), _
                         oReportType.ReportName


'You could also use oReportTypes.ReportInstances
            Set oReportInstances = _
                       REService.CreateServiceObject(bbsoReportInstances)
            oReportInstances.Init REApplication.SessionContext, _
                                             oReportType.ReportID, False, False

            For Each oReportInstance In oReportInstances


With oReportInstance
                    TreeView1.Nodes.Add "Type" & Str$(oReportType.ReportID), _
                       tvwChild, _
                       Str$(.Property(ReR_Property_ReportParameterNamesID)), _
                       .Property(ReR_Property_Name)

                       .CloseDown
                End With
            Next oReportInstance
        Next oReportType
    Next oReportCategory
End Sub
```

Use Report objects to display parameters for the report instance an end-user selects so she can change parameters. She can print, print preview, or save changes (or select any other option available) in *Reports*.

When the parameter for displays, an end-user can do anything she would normally do in ***The Raiser's Edge*** without additional code.

```
Private Sub TreeView1_DblClick()

    Dim lKey As Long

    'This makes sure that they have chosen an
     '    Instance and not a Type or Category
    If Left$(TreeView1.SelectedItem.Key, 8) = "Instance" Then

        Dim oReportInstance As IBBReportInstance

        'This uses the Key from the parent (the Report Type) to specify what Type
         '    of report this is.

        lKey = Int(Mid$(TreeView1.SelectedItem.Parent.Key, 5))

        Set oReportInstance = REService.CreateReportInstance(lKey)
        oReportInstance.Init REApplication.SessionContext

         'This uses the Key from the Instance to Load the correct parameter file
        oReportInstance.Load Int(Mid$(TreeView1.SelectedItem.Key, 9))

        'This displays the Parameter form, at this point the user can do
         '    anything available in the Raiser's Edge.
        oReportInstance.Process ReR_ProcessOption_ShowParameterForm, False, Me
        'At this point, we no longer need to access oReportInstance so we
         '    call CloseDown, it will not be able to close but will close as
         '    soon as the user closes the parameter form or exits the app.
        oReportInstance.CloseDown

    End If

End Sub
```

Code Tables Server

In ***The Raiser's Edge***, a code table is a list of acceptable values for a particular data field. The end-user must select from the list or if he has security, add a new entry to the list. Code tables are used extensively throughout ***The Raiser's Edge***. The end-user's ability to select an entry from a specific list of options simplifies data entry, minimizes typing, and helps to maintain consistency in data entry. You can reduce the size of the database by storing the number that relates to the table entry rather than the actual text. The CodeTablesServer object provides many methods that make using code tables much easier.

First, create an instance of the CodeTablesServer by using the REServices object. Next, call the Init method and provide the SessionContext. Once you have done this, you can use any of the object's methods and a collection of CodeTable objects that contain information about all code tables in the program. Because there may be many opportunities to use CodeTablesServer, you may want to place this initialization code in the Form_Load. When you no longer need the object, call the CloseDown method. You can place this in the Form_Unload.

The LoadCombo method in the CodeTablesServer is a simple way to load a *Visual Basic* combo box with the entries for a particular code table. The following table shows the parameters for this method and the following "Code Sample" on page 58 is an example using this method.

Parameter	Variable Type	Description
oCombo	Object	This is the combo box you want to load.
lTableNumber	ECodeTableNumbers	This is an Enum of all of code tables available in ***The Raiser's Edge***.
bUseShort	Boolean	Optional: Some code tables have short, long, or both descriptions. Normally, the long description is used. However, if you need to use the short description, set this to True. False is the default.
bActiveOnly	Boolean	Optional: In ***The Raiser's Edge***, you can mark table entries Inactive if they are not likely to be used anymore. If this is set to True (which is the default), only entries that are not flagged as Inactive appear.
bClearCombo	Boolean	Optional: If set to True (which is the default), any entries in the combo box are removed before the combo loads.

Code Sample

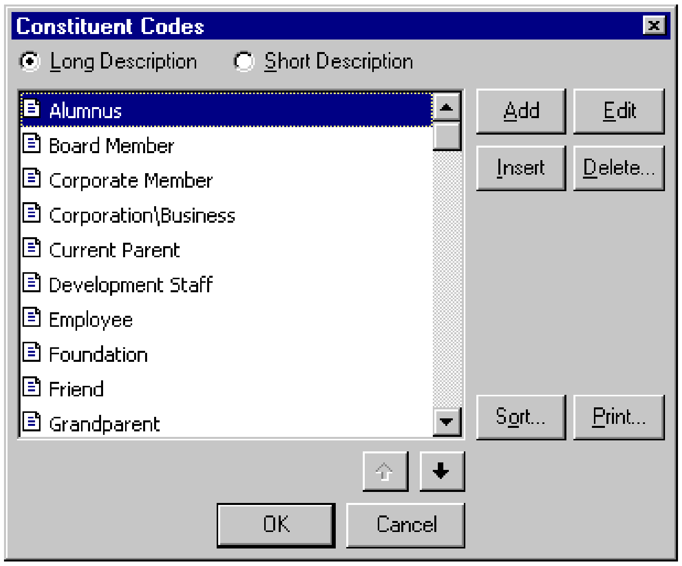
```
Option Explicit

Private moREService As REServices
Private moCodeTablesServer As CCodeTablesServer

Private Sub Form_Load()

     'This is the class that we use to create the service objects
    Set moREService = New REServices
    moREService.Init REApplication.SessionContext

     'This creates an instance of the CodeTableServer
    Set moCodeTablesServer = REService.CreateServiceObject(bbsoCodeTablesServer)
    moCodeTablesServer.Init REApplication.SessionContext

     'This loads the combo with the entries from the Marital Status table
    moCodeTablesServer.LoadCombo Combo1, tbnumMaritalStatus, False, True, True

End Sub

Private Sub Form_Unload(Cancel As Integer)

    moCodeTablesServer.CloseDown
    set moCodeTablesServer = Nothing

End Sub
```

If you provide database ID, you can use the GetTableEntryDescription method to get the table entry description. If you provide the table entry description, use the GetTableEntryID to obtain table entry IDs.

```
Dim lLong As Long
Dim sString As String

'lLong will equal the database ID for the entry
' "Single" in the MaritalStatus table. However this number
' will vary from database to database.
lLong = oCodeTablesServer.GetTableEntryID("Single", tbnumMaritalStatus, False)

' sString will equal "Single"
sString = oCodeTablesServer.GetTableEntryDescription(lLong, tbnumMaritalStatus, False)
```

Table Lookup Handler

The TableLookupHandler object works together with the "Code Tables Server" on page 56 to provide the code table functionality present in ***The Raiser's Edge***. With the TableLookupHandler, you can add a new entry to a code table and display the table entry maintenance form. This form allows the end-user to add, delete, or reorder code table entries.



Use the REService object's CreateServiceObject method to create an instance of the object. Call the Init method. Other than providing the usual SessionContext, you can also provide a reference to an existing CodeTablesServer object. This is not required, but if provided, speeds up the initialization process. As with the CodeTablesServer, it is best to place this in your Form_Load, so that it is available throughout the form. The CloseDown method can be placed in the Form_Unload to release all resources when you are finished.

To display the maintenance form so that an end-user can to select, add, delete, and sort table entries, call the ShowForm method.

Parameter	Variable Type	Description
lCodeTableID	ECodeTableNumbers	This is the ID for the particular code table you want to display.
lFindItemData	Long	Optional: This is the database ID for the table entry you want to have focus when the form displays.
oFormToCenterOn	Object	Optional: This is a reference to the maintenance form you want to display.

Before you display this form, you can set two properties that influence how the form displays. If the ReadOnly property is set to True, the end-user is not able to use the form to add, delete or sort the table entries. If the ShowInactiveEntries property is set to True, the Inactive table entries are included on the form. The Canceled property returns a boolean telling you if the end-user selects to cancel the form. The SelectedItem property returns the database ID of the table entry the end-user selected. If no item is selected, it returns a 0. If an error occurs, the property returns -1. In ***The Raiser's Edge***, if an end-user double-clicks the Label for a table entry field, the maintenance form displays. The following "Code Sample" on page 60 shows an example of how this functionality might be implemented.

Code Sample

```
Option Explicit

Private moCodeTablesServer As CCodeTablesServer
Private moTableLookupHandler As CTableLookupHandler

Private Sub Form_Load()

    'Since the TableLookupHandler uses a CodeTablesServer object,
    '    we can create it first.
    Set moCodeTablesServer = REService.CreateServiceObject (bbsoCodeTablesServer)
    moCodeTablesServer.Init REApplication.SessionContext

    Set moTableLookupHandler = REService.CreateServiceObject(bbsoTableLookupServer)
    'We pass the reference to oCodeTablesServer to speed the Init process.
    moTableLookupHandler.Init REApplication.SessionContext, moCodeTablesServer

End Sub

Private Sub Label1_DblClick()

    moTableLookupHandler.ReadOnly = True
    moTableLookupHandler.ShowInactiveEntries = True

    'By setting sFindItemData, if there is already a table entry in
    '    the combo box, that entry will have focus, when the form is displayed.
    moTableLookupHandler.ShowForm tbnumMaritalStatus, _
        moCodeTablesServer.GetTableEntryID(Combo1.Text, tbnumMaritalStatus), Me

    'If the user cancels the maintenance form then we don't want to change
    '    what is already in the combo box.
    If Not moTableLookupHandler.Canceled Then
        'This uses the SelectedItem property to fill in the Combo box.
        Combo1.Text = moCodeTablesServer.GetTableEntryDescription _
                      (moTableLookupHandler.SelectedItem, tbnumMaritalStatus, False)
    End If

End Sub
```

With the TableLookupHandler object, you can add new table entries to the table throughout the program by using the AddEntry method. When this method is called, the new entry is immediately added to the database. The following table shows the parameters for this method and the following “Code Sample” on page 62 shows an example.

Parameter	Variable Type	Description
bAddOnTheFly	Boolean	This should be set to True so the new table entry immediately adds to the database.

Parameter	Variable Type	Description
lCodeTableID	Long	Optional: This is the code table number for which the table entry belongs. If this is not specified, the current code table set within TableLookupHandler is used.
sShortDescription	String	Optional: This is the short description for this table entry.
sLongDescription	String	Optional: This is the long description for this table entry.
oForm	Object	Optional: The AddEntry method calls the specified object’s Refresh method.

Code Sample

```
Private Sub Combo1_LostFocus()

    Dim sMsg as String

    If Len(Combo1.Text) > 0 Then

        With oCodeTablesServer

            'GetTableEntryID will return a 0 if the current text is not in
             '  the table.
            If .GetTableEntryID(Combo1.Text, tbnumMaritalStatus, False) = 0 Then

                sMsg = "Do you want to add '" & Combo1.Text & "' to the " & _
                                      .TABLENAME(tbnumMaritalStatus) & " table?"

                If MsgBox(sMsg, vbQuestion + vbYesNo) = vbYes Then
                    'This adds the current text to the database and
                      '    Refreshes Combo1. If the AddEntry is unsuccessful
                      '    this will return False.
                    If Not oTableLookupHandler.AddEntry(True, _
                                tbnumMaritalStatus, , Combo1.Text, combo1) Then
                        MsgBox "Unable to add entry", vbInformation + vbOKOnly
                    End If

                Else
                    'If they don't want to add to the table, then they need to
                      '    pick something that is already on the list.
                    Combo1.SetFocus

                End If

            End If

        End With

    End If

End Sub
```

Attribute Type Server

The AttributeTypeServer object provides access to a collection of methods used to gather information about any of the attributes in ***The Raiser's Edge***. You can then use this information to use attributes on your custom forms. Remember, depending on the design of your program, you may be able to use the "Attributes Grid" on page 41 to display your attributes. Attributes in ***The Raiser's Edge*** consist of a **Category**, **Description**, **Date**, and **Comment**. When you create the attribute, the type of information contained in the **Description** is also defined. The **Description** can be a date, a fuzzy date (an incomplete date), text, boolean, currency, number, a constituent name, or a table. If the **Description** type is a table, you may want to use the "Code Tables Server" on page 56 and "Table Lookup Handler" on page 59.

First, use the REServices object to create a new instance of the AttributeTypeServer. After you create the object, call the Init method, passing a valid SessionContext. As with the other service objects, we recommend you place this in the Form_Load so these methods are available throughout your form. You must also call the CloseDown method when you finish using the object, preferably in the Form_Unload. Once the object is properly initialized, you can begin to use the object to gather information about attributes.

The GetAttributeTypeID method requires 2 parameters. The first parameter is a String which is the attribute **Category** in ***The Raiser's Edge***. The second is an Enum of the different kinds of attributes (for example, Action, Fund, Package). The method returns a Long that is the database ID for this particular attribute. Once you know the attribute ID, you can use that ID to find out more information about the attribute. The opposite of this function is the GetAttributeTypeDescription. If you pass the attribute ID, it returns the attribute category as a String. Using the attribute ID, you can use the GetAttributeDataType method to find out what type of data is required for the **Description** of a particular attribute. This method returns a number that corresponds to a member of the bbAttributeTypes enum. The GetAttributeDataType method also accepts a boolean variable that is passed by reference, bUniqueRequirement. After the method is called, the variable sets to True if this attribute type allows only one attribute of this type per record.

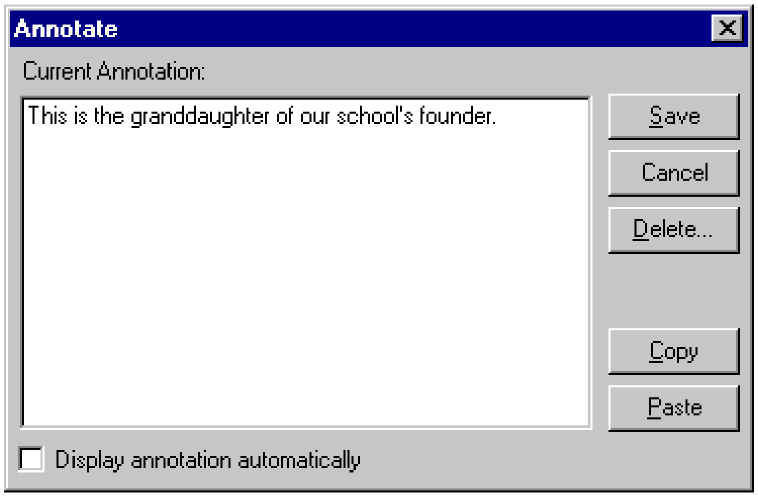
If the data type for the attribute is a table, you may need to get the code table ID for the table. With this, you use the "Code Tables Server" on page 56 and "Table Lookup Handler" on page 59 to simplify your coding. When the GetAttributeCodeTableID method passes through the attribute ID, it returns the code table ID for the table.

In this code sample, a label with the attribute category (in this case “Special Mailing Info”) and either a combo box (if the attribute data type is table or boolean) or a text box (for all other data types) is displayed.

```
Option Explicit

Private moCodeTablesServer As CCodeTablesServer
Private moAttributeTypeServer As CAttributeTypeServer


Private Sub Form_Load()

    Dim lAttribute_ID As Long
    Dim bOnlyOneAllowed As Boolean

    REService.CreateServiceObject (bbsoCodeTablesServer)
    Set moCodeTablesServer = New CCodeTablesServer
    moCodeTablesServer.Init REApplication.SessionContext

    REService.CreateServiceObject (bbsoAttributeTypeServer)
    Set moAttributeTypeServer = New CAttributeTypeServer
    moAttributeTypeServer.Init REApplication.SessionContext

    With moAttributeTypeServer

        lAttribute_ID = .GetAttributeTypeID("Special Mailing Info", _
                                            bbAttributeRecordType_CONSTIT_ADDRESS)

        Label1.Caption = .GetAttributeTypeDescription(lAttribute_ID)

        'bOnlyOneAllowed will now be True or False depending on if this
         '    Attribute is allowed to be present more than once per record
        Select Case .GetAttributeDataType(lAttribute_ID, bOnlyOneAllowed)
             'If the Data Type is Boolean than we add Yes and No to the Combo box
            Case bbAttribute_BOOLEAN
                Combo1.Visible = True
                Combo1.AddItem "Yes"
                Combo1.AddItem "No"

            Case bbAttribute_TABLEENTRY
                Combo1.Visible = True
                'This uses the CodeTablesServer to the load the combo
                  '    with all of the table entries
                moCodeTablesServer.LoadCombo Combo1, _
                               .GetAttributeCodeTableID(lAttribute_ID), , True

            Case Else
                Text1.Visible = True

        End Select

    End With

End Sub
```

Annotation Form

In ***The Raiser's Edge***, end-users can annotate any of the top-level data objects. An annotation is a note that is attached to each record. The end-user can select to have the note display when that record is loaded.



This is the granddaughter of our school's founder.

By using the Annotation Form service object, you can easily add this functionality to your custom applications.

Using the Annotation Form Object

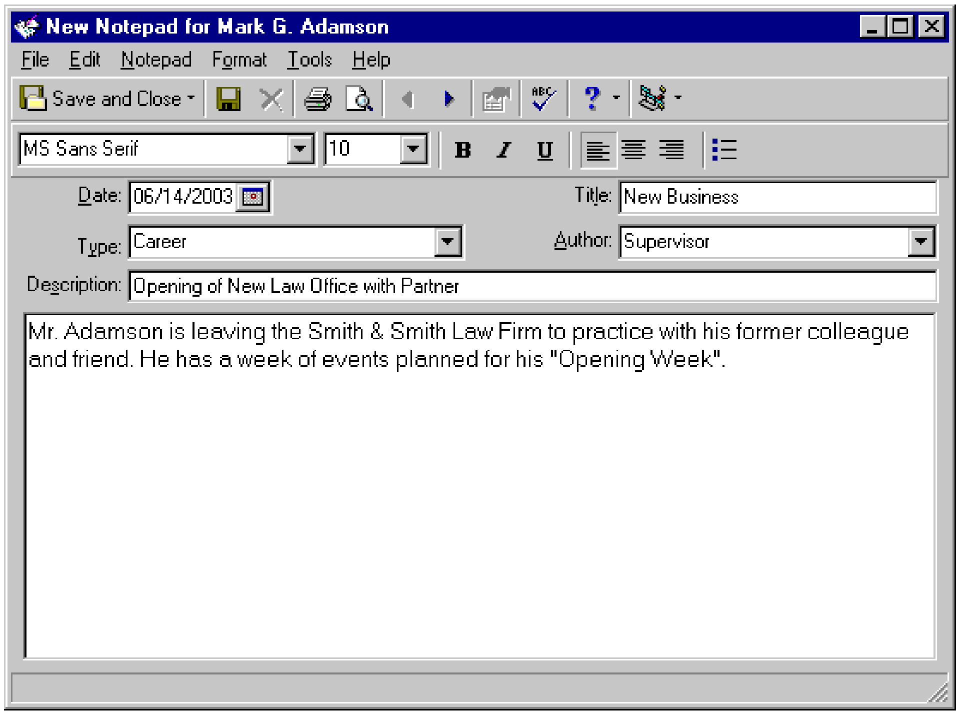
1. Declare an object reference for the REServices object. Create a new instance of the REServices object and use the Init method (remember to call the CloseDown method when finished), passing a valid SessionContext to initialize it.
2. Declare an object reference for the CAnnotationForm object. Create a new instance of the CAnnotationForm object.
3. Set the CAnnotationForm object equal to the REServices.CreateServiceObject method, passing the enum constant bbsoAnnotationForm.
4. Initialize the CAnnotationForm object, using the Init method (remember to call the CloseDown method when finished), passing a valid SessionContext.

To display the form, call the ShowAnnotationForm method, passing the data object the annotation will be attached to. If the data object that is passed does not support an Annotation form (for example, it is not a top-level object) a trappable error is raised. You also must pass the form for which you would like the Annotation form to display. This parameter is optional and if nothing passes, the form displays in the center of the screen. The Annotation form displays modally and the end-user has the same options available in ***The Raiser's Edge***. After you are finished using any Annotation forms in your project, call the CloseDown method to release all the resources being used by the process. It is very important to remember that if the end-user edits the annotation and clicks **Save** on the form, the new text is not saved to the database until you call the Save method for the data object. The following is a code sample of this.

```
Dim REService As REServices
    Set REService = New REServices
    REService.Init REApplication.SessionContext

    Dim oAnnotationForm As CAnnotationForm
    Set oAnnotationForm = REService.CreateServiceObject(bbsoAnnotationForm)
    oAnnotationForm.Init REApplication.SessionContext

    Dim oRecord As CRecord
    Set oRecord = New CRecord

    oRecord.Init REApplication.SessionContext

    oRecord.LoadByField uf_Record_CONSTITUENT_ID, 6

    oAnnotationForm.ShowAnnotationForm oRecord, Nothing

    'Any changes that the user made on the Annotation Form
     '     are not saved until this is called.
    oRecord.Save

     'Clean up.
    oRecord.CloseDown
    Set oRecord = Nothing

    oAnnotationForm.CloseDown
    Set oAnnotationForm = Nothing
```

Notepad Form

In ***The Raiser's Edge***, three of the top-level data objects, Constituent, Gift, and Event allow the end-user to enter multiple notepads for each record. These notepads all add via a common form. The Notepad form service object provides the opportunity to incorporate this functionality into your programs.


New Notepad for Mark G. Adamson
File Edit Notepad Format Tools Help
Save and Close
MS Sans Serif
10

Date: 06/14/2003
Title: New Business
Type: Career
Author: Supervisor
Description: Opening of New Law Office with Partner

Mr. Adamson is leaving the Smith & Smith Law Firm to practice with his former colleague and friend. He has a week of events planned for his "Opening Week".

Using the Notepad Form Object

1. Declare an object reference for the REServices object. Create a new instance of the REServices object and use the Init method (remember to call the CloseDown method when finished), passing a valid SessionContext to initialize it.
2. Declare an object reference for the CNotepad object. Create a new instance of the CNotepad object.
3. Set the CNotepad object equal to the REServices.CreateServiceObject method, passing the enum constant bbsoNotepadForm.
4. Initialize the CNotepad object, using the Init method (remember to call the CloseDown method when finished), passing a valid SessionContext.

Before you use the Notepad Form object, you must first set the NotepadObjects property to a valid collection of Constituent, Gift or Event Notepads. If you want to display an existing Note then you can set the NotepadObjectID equal to the database ID of the Note you want to display. If this property is not set to anything, a blank form displays and new Note is created when the record saves. One area where the Notepad Form differs from other UI forms is that you can select what appears on the caption of the form. This is accomplished by setting the FormCaption property equal to the string you want to appear. When the form displays the caption will be "Notepad For" concatenated with the string you passed.

Parameter	Variable Type	Description
oFormToCenterOn	Object	This is the object for which the Notepad form displays.

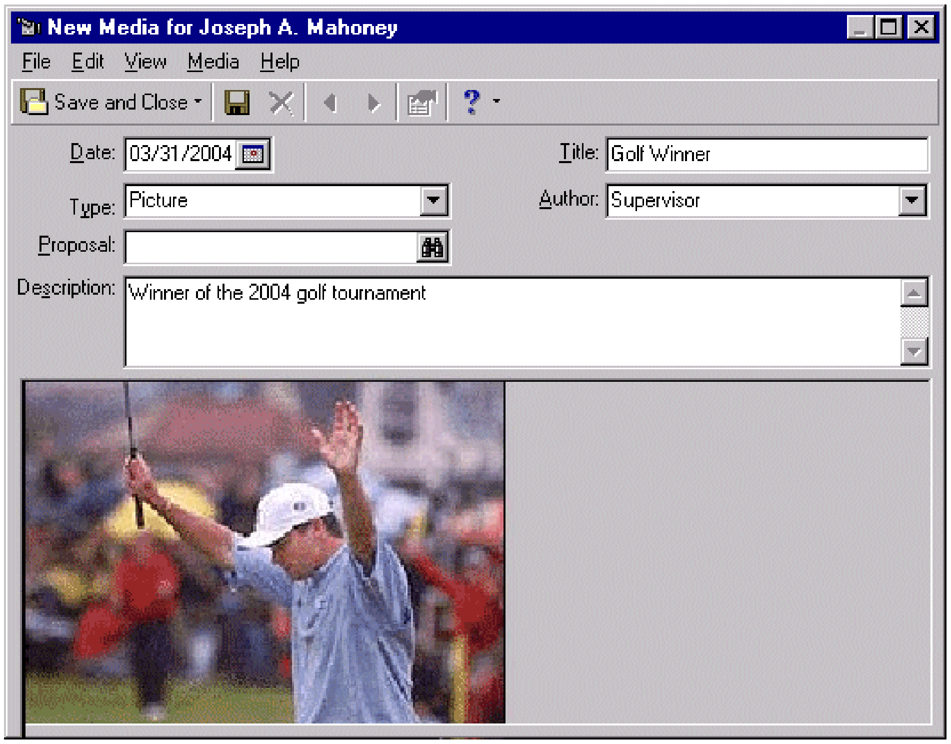
Parameter	Variable Type	Description
oMoveServer	IBBMoveServer	Optional: This establishes how the “VCR” buttons on the form function. This is covered in detail in Programming Reference.
bOKCancel	Boolean	Optional: If set to True (False is the default), the only options under the **File** menu are: **Save**, **Save & Close**, **Properties**, and **Close**.
bViewOnly	Boolean	Optional: If this is set to True (False is the default), the user is able to view the Notepad information, but not edit it.

To display the Notepad form, call the ShowForm method. When an end-user enters all of the data on the form and has selects to save the notepad, the notepad information is not actually saved to the database. To save, call the parent record’s Save method. The following is a code example of how this can be implemented.

```
Dim REService As REServices
    Set REService = New REServices
    REService.Init REApplication.SessionContext

    Dim oRecord As CRecord
    Set oRecord = New CRecord
    oRecord.Init REApplication.SessionContext
    oRecord.LoadByField uf_Record_CONSTITUENT_ID, 6

    Dim oNotepadForm As CNotepadForm
    Set oNotepadForm = REService.CreateServiceObject(bbsoNotepadForm)
    oNotepadForm.Init REApplication.SessionContext

    Set oNotepadForm.NotepadObjects = oRecord.Notepads

    'If this is not set, then a new Notepad will be created.
    oNotepadForm.NotepadObjectID = oRecord.Notepads.Item(1).Fields(NOTEPAD_fld_Id)

    'The caption on the form will read "Notepad for Ms. Julie Marie Bach"
    oNotepadForm.FormCaption = oRecord.Fields(RECORDS_fld_FULL_NAME)

    'In this case, the form will displayed modally, with all File menu
    '    options and allow the user to edit the Note.
    oNotepadForm.ShowForm Nothing

    'The user's changes are not added to the database until this called.
    oRecord.Save

    'Clean Up!
    oRecord.CloseDown
    Set oRecord = Nothing

    oNotepadForm.CloseDown
    Set oNotepadForm = Nothing
```

Media Form

In ***The Raiser's Edge***, you can use the Media tab of a Constituent or an Event record to store various media files, such as documents, bitmaps (graphics), and video files. With the Media form object, you can incorporate this functionality into your custom applications.



Date: 03/31/2004
Title: Golf Winner
Type: Picture
Author: Supervisor
Proposal:
Description: Winner of the 2004 golf tournament

Using the Media Form Object

1. Declare an object reference for the REServices object. Create a new instance of the REServices object and use the Init method (remember to call the CloseDown method when finished), passing a valid SessionContext to initialize it.
2. Declare an object reference for the CMediaForm object. Create a new instance of the CMediaForm object.
3. Set the CMediaForm object equal to the REServices.CreateServiceObject method, passing the enum constant bbsoMediaForm.
4. Initialize the CMediaForm object, using the Init method (remember to call the CloseDown method when finished), passing a valid SessionContext.
5. Set the CMediaForm.MediaObjects property equal to a valid collection of Media objects, such as CRecord.Media.

At this point, the Media form object is ready to be used. If you want to display an existing media item, you can set the MediaObjectID equal to the database ID of the Media item you want to display. If this property is not set to anything, a blank form displays and a new Media item is created when the record saves. One difference in the Media form from other UI forms you have seen is the ability to select what appears on the caption of the form. This is accomplished by setting the NameForCaption property equal to the string you want to appear. When the form displays, the caption is "Media For" concatenated with the string you passed.

To actually display the Media form, you call the ShowForm method. The following table lists the parameters for this method. Once the user completes the form and selects **Save**, the media item information is not saved to the database yet. This is accomplished by calling the parent record’s Save method. The following code sample shows how to implement this.

Parameter	Variable Type	Description
oFormToCenterOn	Object	This is the object over which Media form displays.
oMoveServer	IBBMoveServer	Optional: This establishes how the “VCR” buttons on the form function. This is covered in detail in Programming Reference.

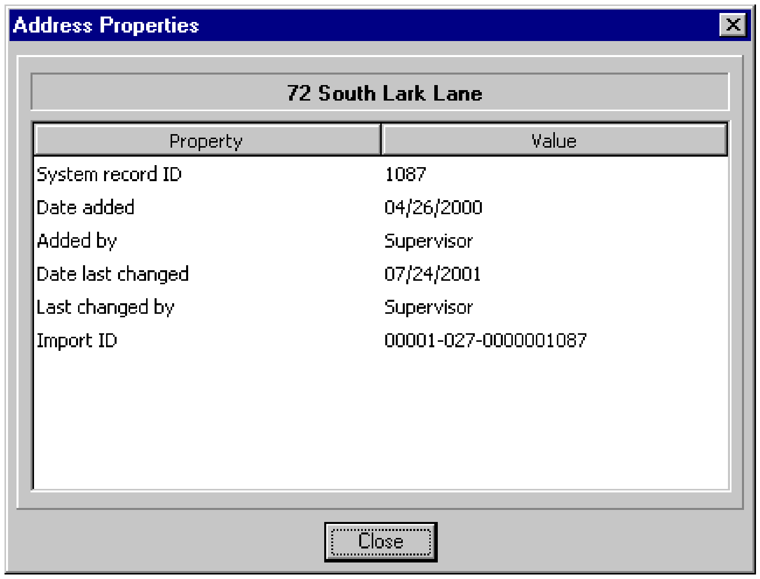
To display the Media form, call the ShowForm method. When an end-user completes the form and selects to save the media item, information is not saved to the database. To save, call the parent record’s Save method.

Code Sample

```
Dim REService As REServices
    Set REService = New REServices
    REService.Init REApplication.SessionContext

Dim oRecord As CRecord
    Set oRecord = New CRecord
    oRecord.Init REApplication.SessionContext
    oRecord.LoadByField uf_Record_CONSTITUENT_ID, 6

    Dim oMediaForm As CMediaForm
    Set oMediaForm = REService.CreateServiceObject(bbsoMediaForm)
    oMediaForm.Init REApplication.SessionContext

Set oMediaForm.MediaObjects = oRecord.Media

     'If this is not set, then a new Media item will be created.
    oMediaForm.MediaObjectID = oRecord.Media.Item(1).Fields(MEDIA_fld_ID)

     'The caption on the form will read "Media for Ms. Julie Marie Bach"
    oMediaForm.NameForCaption = oRecord.Fields(RECORDS_fld_FULL_NAME)

     'In this case, the form is displayed modally.
    oMediaForm.ShowForm Nothing

'The user's changes are not added to the database until this is called.
    oRecord.Save

     'Clean Up!
    oRecord.CloseDown
    Set oRecord = Nothing

    oMediaForm.CloseDown
    Set oMediaForm = Nothing
```

Property Viewer

When using ***The Raiser's Edge***, different statistics are maintained "behind the scenes". For example, when a Gift record is created, the date and the user name of the end-user creating the record are stored in the database. If an end-user wants to see this information, he can select **File, Properties** from the menu bar, and the form displays showing the properties for the particular record type. The fields shown on the Property form vary depending on the record type.



Property	Value
System record ID	1087
Date added	04/26/2000
Added by	Supervisor
Date last changed	07/24/2001
Last changed by	Supervisor
Import ID	00001-027-0000001087

With the service object Property Viewer, you can display this form in your applications.

Using the Property Viewer

1. Declare an object reference for the REServices object. Create a new instance of the REServices object and use the Init method (remember to call the CloseDown method when finished), passing a valid SessionContext to initialize it.
2. Declare an object reference for the IBBPropertyViewer object. Create a new instance of the IBBPropertyViewer object.
3. Set the IBBPropertyViewer object equal to the REServices.CreateServiceObject method, passing the enum constant bbsoPropertyViewer.
4. Initialize the IBBPropertyViewer object, using the Init method (remember to call the CloseDown method when finished), passing a valid SessionContext.

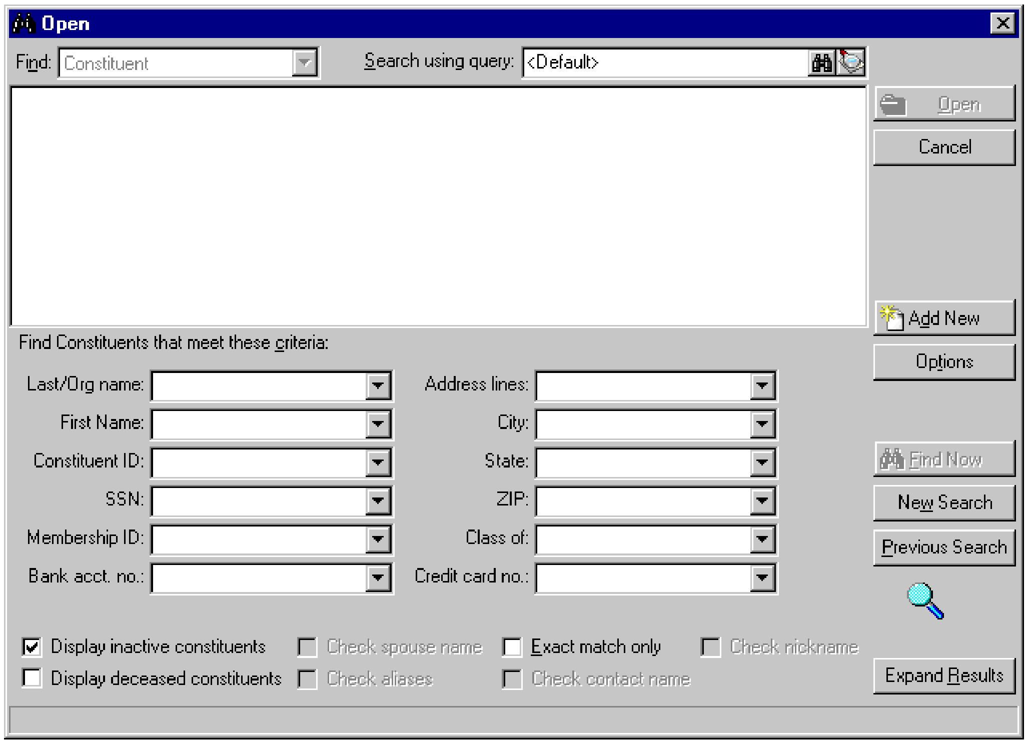
5. Display the Properties form by calling the ShowForm method, passing the data object you want to see the properties for. You can also select the form to center the Properties form.

```
Dim REService As REServices
    Set REService = New REServices
    REService.Init REApplication.SessionContext

    Dim oRecord As CRecord
    Set oRecord = New CRecord

    oRecord.Init REApplication.SessionContext
    oRecord.LoadByField uf_Record_CONSTITUENT_ID, 6

    Dim oPropertyViewer As IBBPropertyViewer
    Set oPropertyViewer = REService.CreateServiceObject(bbsoPropertyViewer)
    oPropertyViewer.Init REApplication.SessionContext

    'This will display the properties for the Preferred Address
     '    for this Constituent.  If the object that you pass doesn't
     '    support Properties, a trappable error will be raised.
    oPropertyViewer.ShowPropertyForm oRecord.PreferredAddress, Me

     'Clean Up!
    oRecord.CloseDown
    Set oRecord = Nothing

    oPropertyViewer.CloseDown
    Set oPropertyViewer = Nothing
```

When you create the Property Viewer, you can use it repeatedly to display the Properties form for any type of record. You may want to place the code that creates a new instance of the IBBPropertyViewer in your Form_Load. Then, you can display the Properties form from anywhere on your custom form.

Search Screen

The search screen (on the UI, this is called the “Open” screen) is used extensively throughout ***The Raiser’s Edge***. For example, it is used any time the user needs to open a specific record or choose a query for a report. With the search screen object, you can incorporate this functionality into your project. The search criteria and filters change automatically based on the type of record, and you can also give your end-users the opportunity to create a new record.



Using the Search Screen Object

1. Declare an object reference for the REServices object. Create a new instance of the REServices object and use the Init method (remember to call the CloseDown method when finished), passing a valid SessionContext to initialize it.
2. Declare an object reference for the IBBSearchScreen object. Create a new instance of the IBBSearchScreen object.
3. Set the IBBSearchScreen object equal to the REServices.CreateServiceObject method, passing the enum constant bbsoSearchScreen.
4. Initialize the IBBSearchScreen object, using the Init method (remember to call the CloseDown method when finished), passing a valid SessionContext.

Because the search screen may be used multiple times in a single project, you may want to declare this as a global or module level reference. You can call the Init when the form loads or when it is first used and then call the CloseDown when the form unloads.

Before you display the search screen, you need to tell it what types of records you want to be available in the **Find** combo frame. Call the AddSearchType method to add at least one search type before displaying the form. If you want to have more than one type of record available, there are two syntax styles supported (see the following code sample). If you have used the object previously, you may want to call the ClearSearchTypes method to clear any existing search types.

```
'Method 1
    oSearchScreen.AddSearchType SEARCH_CAMPAIGN
    oSearchScreen.AddSearchType SEARCH_FUND
    oSearchScreen.AddSearchType SEARCH_APPEAL

'Method 2
    oSearchScreen.AddSearchType SEARCH_CAMPAIGN,SEARCH_FUND,SEARCH_APPEAL
```

Before you display the search screen form, you can set optional properties for the form. If you set the AllowAddNew property to True, when the form displays, there is an **Add New** button present. However, you must write the code that actually creates the new record. If you have added multiple search types, you can also set the DefaultSearchType property to determine the initial search type when the form displays.

When you are ready to display the search screen, call the ShowSearchForm method. This displays the form modally and returns False if the end-user clicks **Cancel**. When the end-user clicks any button that closes the form (**Open**, **Cancel** or **Add New**), you need to find out what he has selected. The SelectedOption property returns the button the end-user selected. The enum bbSearchScreenOption lists options, simplifying your programing. If multiple search types were available, the SelectedSearchType property determines the search type the end-user selected. This returns a value from the enum bbSearchTypes. The SelectedDataObject property returns a reference to the actual data object the end-user selected. The following code shows an example of how you can use the search screen object.

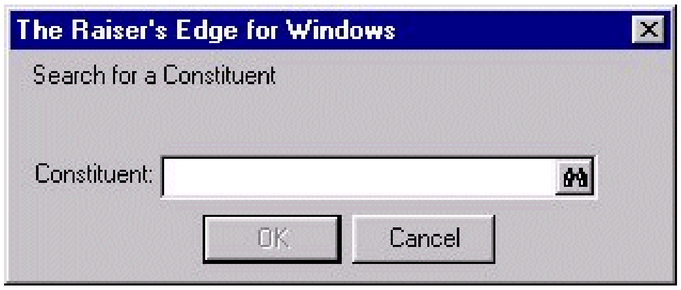
```
Private Sub UsingTheSearchScreen()

    Dim REService As REServices
    Set REService = New REServices
    REService.Init REApplication.SessionContext

    Dim oSearchScreen As IBBSearchScreen
    Set oSearchScreen = REService.CreateServiceObject(bbsoSearchScreen)
    oSearchScreen.Init REApplication.SessionContext


With oSearchScreen
        .ClearSearchTypes
        .AddSearchType SEARCH_CAMPAIGN, SEARCH_FUND, SEARCH_APPEAL
        .DefaultSearchType = SEARCH_CAMPAIGN
        .AllowAddNew = True


If .ShowSearchForm Then
            Select Case .SelectedSearchType
```

```
Case SEARCH_CAMPAIGN
                    Dim oCampaign As CCampaign

                    Dim oCampaignForm As CCampaignForm
                    Set oCampaignForm = New CCampaignForm
                    oCampaignForm.Init REApplication.SessionContext

                    If oSearchScreen.SelectedOption = SRCH_FRM_OPEN Then
                        Set oCampaign = oSearchScreen.SelectedDataObject
                    Else
                        Set oCampaign = New CCampaign
                        oCampaign.Init REApplication.SessionContext
                    End If


Set oCampaignForm.CampaignObject = oCampaign
                    oCampaignForm.ShowForm True, Me, True

                    oCampaignForm.CloseDown
                    Set oCampaignForm = Nothing

                    oCampaign.CloseDown
                    Set oCampaign = Nothing

                Case SEARCH_FUND
                      'Same as above except substitute Fund objects

                Case SEARCH_APPEAL
                      'Same as above except substitute Appeal objects
        End Select

    End If

     'Clean Up!
    oSearchScreen.CloseDown
    Set oSearchScreen = Nothing

    REService.CloseDown
    Set REService = Nothing
End Sub
```

Using the search screen object properly in your custom applications not only simplifies your coding, but also provides a common interface for selecting records to your end-users. Other methods and properties available to use with this form are explained in Programming Reference.

MiscUI

As the name implies, this object gives you access to miscellaneous forms and functions that are helpful in designing your projects in ***The Raiser's Edge***. For example, using the PromptForDataObject method provides your end-user with a simple form for selecting a record.



Using the MiscUI Object

1. Declare an object reference for the REServices object. Create a new instance of the REServices object and use the Init method (remember to call the CloseDown method when finished), passing a valid SessionContext to initialize it.
2. Declare an object reference for the IBBMiscUI object. Create a new instance of the IBBMiscUI object.
3. Set the IBBMiscUI object equal to the REServices.CreateServiceObject method, passing the enum constant bbsoMiscUI.
4. Initialize the IBBMiscUI object, using the Init method (remember to call the CloseDown method when finished), passing a valid SessionContext.

This object has a number of helpful methods, one of which is the PromptForDataObject. This provides a form for your end-user to select a record. The method requires three parameters (see the following table) and returns a long integer that is the database ID of the record that the end-user selected (or a zero if the end-user canceled). If the search string the end-user enters in the textbox is not valid, he is automatically prompted with the standard search screen or prompted to enter the search string information again.

Parameter	Variable Type	Description
lSearchType	bbSearchTypes	This is an enum of the record types available for searching.
sObjectNameCaption	String	This is the string that appears to the left of the search string textbox. It provides your end-user with an idea of the type of record she is searching for.
sDescriptionCaption	String	This is the string that displays just below the title bar on the form.

Another helpful method is the **Quick Find** method. If you provide a search string and the record type you want to search, this method returns True if it was able to find a record with the search string. You can also pass two variables by reference, representing the name and the database ID of the matching record. The following table lists the parameters for this method.

Parameter	Variable Type	Description
lObjectType	bbDataObjConstants	This is an enum of the record types available for searching purposes. *Note:* **Quick Find** is designed to be used with constituents, gifts, campaigns, funds, appeals, events, and banks only.
sSearchString	String	This is the string you are searching for.
sFoundName	String	This is passed By Reference and is set to Name for the record that is found.
lFoundID	Long	This is passed By Reference and is set to the database ID for the record that is found.
bPromptIfNoMatch	Boolean	Optional: If this is set to True (which is the default) a message box provides the end-user with the option to use the search screen again if the search string is not found.

Two methods that work similarly to provide a way to display the UI form for any data object are the ShowUIForDataObject and the ShowUIForDataObjectO methods. ShowUIForDataObject accepts the database ID of the record you want to display, but ShowUIForDataObjectO is passed a reference to the data object you want to show. The following table shows the other parameters for these two methods. These work the same way for both methods.

Parameter	Variable Type	Description
lDataObjectType	bbDataObjConstants	This is an enum of the record types available.
bModal	Boolean	If this is set to True the form is displayed modally.
lCallingHwnd	Long	Optional: You can pass a handle to a form and the method verifies this form is not already loaded before displaying the new form.
oMoveServer	IBBMoveServer	Optional: This establishes how the “VCR” buttons on the form function. This is covered in detail in Programming Reference.
FormToCenterOn	Object	Optional: This is the object for which the UI form is centered.

Code Sample

```
Dim sName As String
    Dim lID As Long

    Dim oRecord As CRecord
    Set oRecord = New CRecord
    oRecord.Init REApplication.SessionContext

    Dim REService As REServices
    Set REService = New REServices
    REService.Init REApplication.SessionContext

    Dim oMiscUI As IBBMiscUI
    Set oMiscUI = REService.CreateServiceObject(bbsoMiscUI)
    oMiscUI.Init REApplication.SessionContext

     'This uses Text1.Text as the search string for a constituent.
    If oMiscUI.QuickFind(bbdataConstituent, Text1.Text, sName, lID, False) Then
         oRecord.Load lID
    Else
        'This will display a form for the user to search for a constituent.
        IID = oMiscUI.PromptForDataObject(SEARCH_CONSTITUENT, "Constituent", _
         “Search for a Constituent”)
        If lID Then oRecord.Load lID
    End If

    'This will display the UI form for a constituent, if the user has not selected
     '    a valid record by now then a blank form will be displayed.
    oMiscUI.ShowUIForDataObjectO bbdataConstituent, oRecord, True, , , Me

     'Clean Up!
    oRecord.CloseDown
    Set oRecord = Nothing

    REService.CloseDown
    Set REService = Nothing

    oMiscUI.CloseDown
    Set oMiscUI = Nothing
```

These methods provide a simple way to incorporate powerful functions into your custom applications without writing a lot of new code. There are other methods available through the MiscUI object. These are documented in Programming Reference.

Advanced Concepts and Interfaces

Review this section to learn about the IBBDataObjectInterface, a powerful *VBA* tool that supports the use of interfaces. You can also learn about the IBBMetaField interface, which provides a convenient way to find out and change information about the individual data fields used in ***The Raiser’s Edge***. Lastly, using transactions, you can add or remove a number of items from a collection or make changes to the fields in a data object temporarily.

Using the IBBDataObject Interface

One of the powerful features of *Visual Basic* is that it supports the use of interfaces. Interfaces are advanced programming techniques that provide a way for the ***Raiser's Edge*** developers to make the program code more efficient and easier to maintain. An interface is like a contract. Any class that implements a specific interface guarantees the class supports a certain type of behavior. Many objects in ***The Raiser's Edge*** object model implement other interfaces. For example, all data objects, such as CGift, or CEducation, implement the IBBDataObject interface. This provides a way for you to refer to any data object that implements IBBDataObject in a generic fashion. Referring to the IBBDataObject interface gives you access to some properties and methods that are not available when referring to the actual class.

By referring directly to the IBBDataObject interface, you can use the Initialized property to find out if an object has been initialized. You can use the Dirty property to determine if an object has had changes made to it since it was saved.

Another important property of the IBBDataObject interface is the ObjectName property. This returns the name of the class of object you are using. One of the most important properties is the MetaField property. This provides access to information about the types of data that any field expects and other important information. The below code sample shows how to use interfaces to generically refer to objects. Notice that it includes a reference to the IBBTopObject interface. All top-level objects implement this interface. Using this interface provides a way to refer to and access the methods and properties common to all top level objects.

```
Private Sub UsingInterfaces(oTopObject As IBBTopObject)

    Dim oIBBDataObject As IBBDataObject
    Set oIBBDataObject = oTopObject

    'This makes sure that the object passed in has been initialized
    If Not oIBBDataObject.Initialized Then
        oTopObject.Init REApplication.SessionContext
    End If

    'Allows you to find out what class has been passed
    Select Case oIBBDataObject.OBJECTNAME
        Case "CAction"
            oIBBDataObject.Fields(ACTION_fld_NOTES) = "Test Note"
        Case "CGift"
            oIBBDataObject.Fields(GIFT_fld_Reference) = "Test Note"
        Case "CRecord"
            oIBBDataObject.Fields(RECORDS_fld_BIRTHPLACE) = "North Carolina"
    End Select

    If oIBBDataObject.Dirty Then oTopObject.Save

    'Clean Up!
    Set oIBBDataObject = Nothing

End Sub
```

Using the IBBMetaField Interface

The IBBMetaField interface provides a convenient way to find out and change information about the individual data fields used in ***The Raiser's Edge***. For example, in designing your programs, you may need to know if a particular field has been defined as required through *Configuration* in ***The Raiser's Edge*** or you may need to know the type of data a specific field requires (such as date, number, or percentage).

Every data object in ***The Raiser's Edge*** object model provides an IBBMetaField interface to determine this information. Each data object has a Fields property, array of the actual data in each field, and most of the properties in the IBBMetaField interface return a similarly numbered array. This makes using the IBBMetaData and the actual data object together much simpler. The following table displays the most commonly used properties and what they represent.

Parameter	Variable Type	Description
DisplayText	String	This returns the user-defined Display As text for this field.
FormatDescription	EFormatDescriptors	This returns the type of data contained in this field.
Required	Boolean	If returns True if the field is required.
UserRequired	Boolean	If returns True if this is a field the user has selected to make required.
FormToCenterOn	Object	Optional: This is the object for which the UI form is centered.

You can set some of the properties through your program. For example, DisplayText, UserHidden, and UserRequired can be changed. The Save method saves those changes to the database. The following code shows an example of how you might use these properties to load an array of textboxes and accompanying labels and also allow your end-user to change the Display Text by double-clicking on the label. Notice we use the “Using the IBBDataObject Interface” on page 80 interface to return a reference to the IBBMetaField interface.

Code Sample

```
Option Explicit

Private moAction As CAction
Private moMetaField As IBBMetaField
Private moDataObject as IBBDataObject

Private Sub Form_Load()

    Dim i As Integer

    Set moAction = New CAction
    moAction.Init REApplication.SessionContext

     'Provides access to the IBBMetaField interface for the moAction object
    Set moDataObject = moAction
    Set moMetaField = moDataObject.MetaField


For i = 1 to moMetaField.Count

         'If the field should be hidden we want to skip it.
        If Not moMetaField.UserHidden(i) Then
             'This is determined in Configuration.
            Label1(i).Caption = moMetaField.DisplayText(i)

            'You need to check if it is system-required or
             '    if it is user-required.
            If moMetaField.Required(i) Or moMetaField.UserRequired(i) Then
                Label1(i).ForeColor = vbRed
            End If


If moMetaField.FormatDescriptor(i) = fmtAMOUNT Then
                Text1(i).Text = "$" & Text1(i).Text
            End If
        End If
    Next i
End Sub

Private Sub Label1_DblClick(Index As Integer)

    Dim s As String
    s = InputBox("Enter the new Display Text")


If Len(s) Then
        moMetaField.DisplayText(Index) = s
         'This must be called to save the user's changes.
        moMetaField.Save
    End If
End Sub
```

```
Private Sub Form_Unload(Cancel As Integer)

     'Clean up!
    Set moMetaField = Nothing
    set moDataObject = Nothing
    moAction.CloseDown
    Set moAction = Nothing
End Sub
```

Transactions

In ***The Raiser's Edge*** object model, there are a number of collections that support the use of transactions. Also, any object that implements IBBDataObject supports transactions. With transactions, you can add or remove any number of items from a collection or make changes to the fields in a data object temporarily until you decide to permanently make those changes. At any point in a transaction, you can select to undo the changes you made. Three methods and one property work together to provide the transactions functionality.

The BeginCollectionTransaction (or BeginFieldsTransaction for data objects) method signals the collection or object you want to begin a transaction at this point. If later you select to undo the changes, the collection or object returns to exactly the same state it is in at this time.

Calling the CommitCollectionTransaction (or CommitFieldsTransaction for data objects) method tells the collection or object you are finished with this transaction, and you want to make any changes made to the collection or object since the BeginCollectionTransaction became permanent. However, this does not make any changes to the database itself; those changes can be made only by calling the parent record's Save method.

To return the collection or the object to the state it was in when you started the transaction, call the RollbackCollectionTransaction (or RollbackFieldsTransaction for data objects) method. After calling this method, the collection or object is exactly as it was when the transaction began.

When using collections, you can use the InTransaction property to check to see if the collection is in the middle of a transaction. This is important because if you call CommitCollectionTransaction or RollbackCollectionTransaction when there is no active transaction, an error is raised. The following code sample is an example of how these can be used.

```
Dim oRecord As CRecord
    Set oRecord = New CRecord

    oRecord.Init REApplication.SessionContext
    oRecord.LoadByField uf_Record_CONSTITUENT_ID, 3

    Dim oIndividuals As CIndividuals
    Set oIndividuals = oRecord.Relations.Individuals

    oIndividuals.BeginCollectionTransaction
    oIndividuals.Remove oIndividuals.Item(1), False

     'If we just removed the last Individual relation then Rollback.
    If oIndividuals.Count = 0 Then
        oIndividuals.RollbackCollectionTransaction
    Else
        oIndividuals.CommitCollectionTransaction
    End If

     'Any changes to the collection are not saved to the database until now.
    oRecord.Save
    Set oIndividuals = Nothing

    oRecord.CloseDown
    Set oRecord = Nothing
```

Custom View: Creating Custom Parts

With *Custom View* for ***The Raiser's Edge***, you can create a customized view of individual and organization constituent records. This makes it possible to view records showing only the information you want to see. By creating these customized screens, you can select and view only what is important and functional for your organization. Every field available on a constituent record may not be useful to you. Using *Custom View*, you can easily select and organize only the fields and items you want to appear for a constituent record.

Saving custom view parts is a key element in using *Custom View*. When you create custom views, you can incorporate saved parts from existing custom views. Using this "building block" method helps you make your custom views more consistent and eliminates duplication of effort from creating the same part over and over again on different views.

Custom Parts

VBA enables you to go a step further. In addition to creating saved parts, you can use *VBA* to create custom parts for your custom views. Custom parts are similar to saved parts, yet give you unlimited flexibility to create and customize parts of custom screens to fit the particular needs of your organization. While saved parts can incorporate any of the functionality included in Custom View, your custom parts can include any feature you wish to create. Once you create a custom part, you can make it available for all your users to include in custom views they create.

Standard custom views are read-only. However, you can use *VBA* to create custom parts that enable your users to edit constituent information when viewing constituent records in custom view mode.

Adding a Custom Part

To create custom parts in *VBA*, you must perform several steps. You must create a system macro that creates an instance of the custom part class you will create. Then you must create the class for your custom part. After you perform these two steps, you can click the **Custom View** link in *Configuration* to use the custom part when you create a custom view.

To use the sample below, it is very important that you first set a reference to the Microsoft HTML Object Library. From *VBA*, select **Tools**, **References**, and mark the **Microsoft HTML Object Library** checkbox. Click **OK**.

Then, add a system_macro with IBBCustomView in the signature to call the class you will create for your custom part. In this example, our class will be named “CHelloWorldPart”.

```
' This public sub is required in for the custom view designer to discover that
' we have a custom part available.

' 1. Any custom view must have a corresponding sub with the same signature as this
sample.

' 2. You must set oView to a new instance of the class you created that implements
' the IBBCustomView interface.


Public Sub VBACustomView(ByRef oView As IBBCustomView)
    Set oView = New CHelloWorldPart
End Sub
```

Now create the class module for the custom view. To add the class, in the **Project - System** box, right-click **System** and select **Insert**, **Class Module**. Paste in the sample code below and name the class module CHelloWorldPart.cls. This sample demonstrates a custom part that will display the message “Hello World” a text link over a background color you specify. When a user clicks the text link, the custom part calculates the total amount of cash gifts donated by a constituent.

You must place your implementation (both design-time and runtime) in the class instantiated in the system macro you created. The design-time implementation can contain information to help custom view users when they use the custom part while creating a custom view. For example, you may want to explain that the calculation of cash gifts can be performed only at runtime. The runtime implementation contains the actual functionality of the custom part when the end user views a constituent record in a custom view that incorporates it.

Because Custom View uses HTML as its rendering engine, it is helpful to be familiar with HTML Styles, Dynamic HTML, and Cascading Style Sheets. Knowledge of XML can also be very helpful in creating custom parts.

```
Option Explicit

Implements IBBCustomView
Implements IBBCustomViewDesign
Implements IBBCustomViewEvents

Private moHost As IBBCustomViewHost
Private moRec As CRecord
Private WithEvents moDIV As HTMLDivElement      ' Withevents so we can catch user
                                                    ' events on our custom part such as
                                                     ' "click"

Private WithEvents moCashLink As HTMLSpanElement

' This method is called when an end user opens a constituent record
' and selects this custom view
Private Sub IBBCustomView_Init(ByVal oHTMLContainerElement As MSHTML.IHTMLElement, _
ByVal oHost As BBREAPI7.IBBCustomViewHost, ByVal sParameterString As String)

    ' Cache references to key objects
    Set moHost = oHost
    ' This is the data object representing the current constituent record
    Set moRec = moHost.CurrentDataObject

    ' Custom View uses HTML as its rendering engine, so we need to be familiar
    ' with dynamic HTML/CSS/Styles.  This line gives us programmatic access
    ' to the DIV element that is encapsulating our custom part

    ' Note: Remember to set a reference to the HTML Object Library so we will have
    ' early-bound access to HTML elements
    Set moDIV = oHTMLContainerElement

    ' Example: using a style property to change the background color of our part
    moDIV.Style.backgroundColor = "aqua"

    ' Call the routine that will handle rendering our custom part's content
    renderCustomPart

End Sub

Private Sub IBBCustomView_CloseDown()

    ' Called when end user closes the constituent record,
    ' good time to do clean-up
    Set moDIV = Nothing
    Set moRec = Nothing
    Set moHost = Nothing

End Sub
```

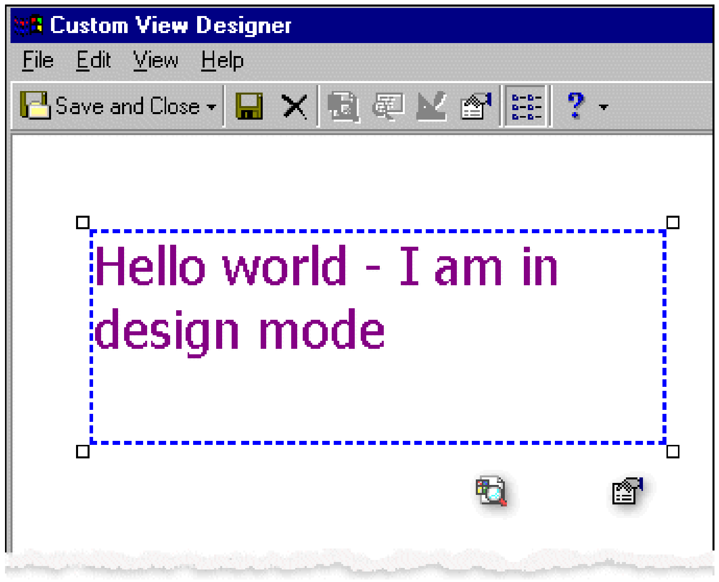
```
' This method is invoked only when the end user is interacting with the
' custom view in design-mode (that is, from Configuration)
Private Sub IBBCustomViewDesign_CloseDown()

     ' Called when end user closes the custom view designer

End Sub

Private Property Get IBBCustomViewDesign_Description() As String
     ' Optionally return a verbose description
    IBBCustomViewDesign_Description = "This description provides additional info"

End Property


Private Property Get IBBCustomViewDesign_HasParameters() As Boolean
     ' Not implemented in The Raiser's Edge 7.5; reserved for future use
End Property

Private Sub IBBCustomViewDesign_Init(ByVal oSC As BBREAPI7.IBBSessionContext)
    ' This method is invoked when a user opens a custom view in which our custom
     ' part is sited in design mode.
End Sub


Private Property Get IBBCustomViewDesign_Name() As String

    ' This is the actual description of the custom view that displays
     ' for our node in the treeview under the custom parts category in the
     ' Additional
     ' Objects box in the Custom View Designer
    IBBCustomViewDesign_Name = "Hello World Custom View Sample"

End Property


Private Sub IBBCustomViewDesign_RenderDesignTimeContent(ByVal oHTMLContainerElement As _
MSHTML.IHTMLElement, ByVal sParameterString As String)

    ' This method is used to display information about our custom part in the designer.
    ' Typically, this will just be some informative text. For example, you can explain that
    ' in
    ' in design mode, the part cannot process information and that this can be done in
    ' runtime
    ' Note the use of dynamic HTML styles.
    oHTMLContainerElement.innerHTML = "<SPAN style='font-size:20pt;_
color:purple'>Hello world - I am in design mode</SPAN>"

End Sub
```

```
Private Function IBBCustomViewDesign_ShowParameterForm(sParameterString As String) _
As Boolean
    ' When a user clicks the "parameter" glyph in the designer, this method is invoked.
    ' Typically, you would show some sort of front-end form to get criteria from the user.
    ' Then, you store the criteria as a string (hint: use XML). The custom view engine will
    ' store this string for you and make it available at runtime (when an end user opens an
    ' actual constituent record). At that point, you would interrogate the string and apply
    ' criteria appropriately.
End Function

Private Sub IBBCustomViewEvents_AfterSave()
    ' This is equivalent to the VBA AfterSave Data Object event
    MsgBox "User just saved record"
End Sub


Private Sub IBBCustomViewEvents_BeforeSave(bcancel As Boolean)
    ' This is equivalent to the VBA BeforeSave data object event
    MsgBox "User about to save record"
End Sub


Private Function IBBCustomViewEvents_onHTMLEvent(ByVal oHTMLEvent As MSHTML.IHTMLEventObj)_
As Boolean
    ' This is a general purpose routine that can be used as a catch-all for all HTML events
    ' that are fired on your custom part. Ideally, you should use direct "WithEvents"
    ' references to
    ' HTML elements in your custom part (see declaration of moDIV variable at the top
    ' of this class), but this can be used as well.
End Function


Private Sub IBBCustomViewEvents_QueryCloseDown(bcancel As Boolean)

    ' If bCancel is set to true, then closing of the form by the end user is
    ' overridden (typically due to some business rule violation)


End Sub


Private Sub IBBCustomViewEvents_QuerySwitchView(bcancel As Boolean)

    ' When multiple custom views are available, the constituent form allows the
    ' end user to cycle between them. If you set bCancel to true here, you
    ' will override the view change and the current view will remain active.
End Sub
```

```
Private Sub renderCustomPart()

    ' Show our initial interface, which is a link that when clicked displays
    ' the total amount of cash this constituent has donated

    ' First inject our "link." We use an HTML SPAN element and set its font to underlined
    ' blue to give the appearance of a link
    moDIV.innerHTML = "Hello world " & "<SPAN ID=cashLink style='text-decoration:_
underline;font-size:9pt;color:blue;cursor:hand'>Click me to calculate total cash</SPAN>"

     ' Now hook up our EventVariable (so we can trap the click)
    Set moCashLink = moDIV.all("cashLink")

    ' At this point, we have a variable (moCashLink) ready to respond to the click event
    ' See the moCashLink_onclick() routine for the code that will be executed when our link
    ' is clicked

End Sub
```

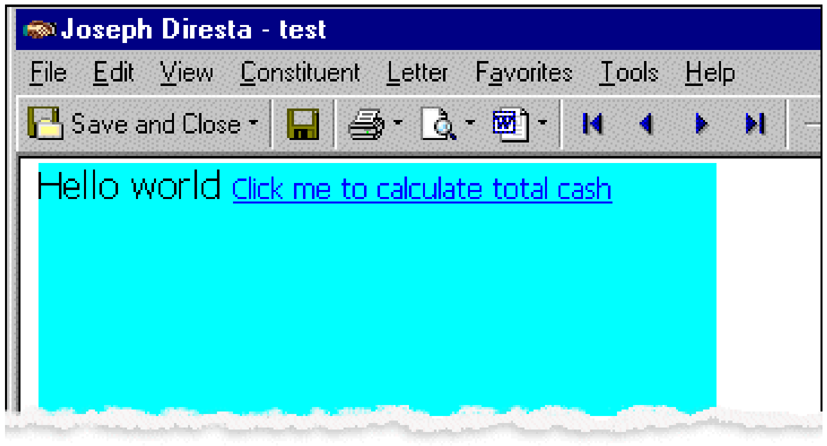
```
Private Function moCashLink_onclick() As Boolean

    ' The user just clicked our link, so update our custom part to display total cash
    showTotalCash

End Function

Private Sub showTotalCash()

     'For our sample we will display the total amount of cash donations given by
    ' this constituent
    Dim oGift As CGift

    Dim cTotal As Currency
    cTotal = 0

    For Each oGift In moRec.Gifts

         ' If it is a cash gift
        If oGift.Fields(GIFT_fld_Type) = "Cash" Then

            cTotal = cTotal + oGift.Fields(GIFT_fld_Amount)

        End If

        oGift.CloseDown

    Next oGift


    moDIV.innerHTML = "<SPAN style='font-size:20pt;font-family:impact;color:green'>_
Total Cash:" & FormatCurrency(cTotal) & "</SPAN>"

End Sub
```

To use the new custom part, click **Configuration** in the Raiser's Edge bar and click the **Custom View** link. In Custom View, custom parts are available from the Fields screen, at the bottom of the **Additional Objects** box. When you drag our sample into a custom view you are creating, here is what you see in design mode.



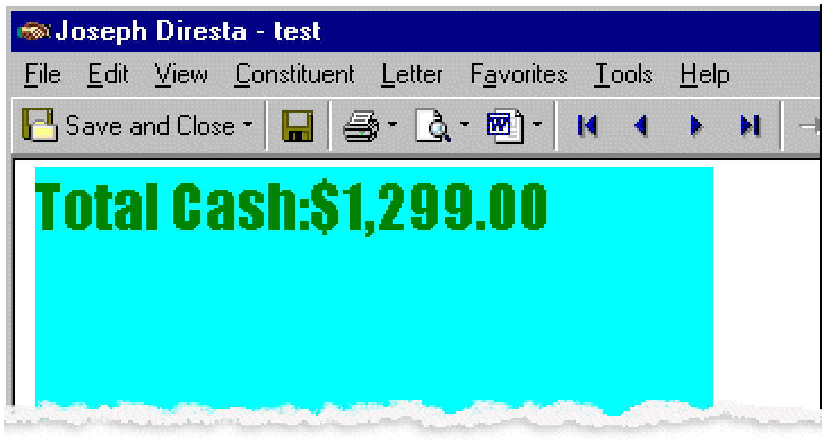
Hello world - I am in design mode

Note the descriptive text you invoked using DHTML styles. When the custom part is added to a custom view called "test", and that custom view is used to view a constituent record, here is how the part appears. Note the specified aqua background and the HTML SPAN element that gives the appearance of a link.



Hello world Click me to calculate total cash

When you click the link, the custom part processes the constituent’s gifts and returns the total value of cash gifts.



Total Cash:$1,299.00

For more information about using the Custom View Designer, see the *Custom View Guide*.

API

Chapter 2

Contents

This chapter introduces *API for Advanced Application Development*. This optional module enables the experienced developer to leverage the power of ***The Raiser's Edge*** programmatically from third party or custom applications.

This section builds upon many concepts introduced in the Essentials chapter, so you should be familiar with the material discussed in that guide before you begin.

Please remember....

We provide programming examples for illustration only, without warranty either expressed or implied, including, but not limited to, the implied warranties of merchantability and/or fitness for a particular purpose. This article assumes that you are familiar with Microsoft *Visual Basic* and the tools used to create and debug procedures. Blackbaud Customer Support can help explain the functionality of a particular procedure, but they will not modify, or assist you with modifying, these examples to provide additional functionality. If you are interested in learning more about ***The Raiser's Edge*** optional modules *VBA* and *API*, contact our Sales department at solutions@blackbaud.com.

What is API?

This section provides a general, high-level overview of *API*.

API vs. VBA

This section explains how *API* differs from ***The Raiser's Edge*** *VBA* support. It is important to understand which of these customization methodologies is appropriate for your task.

API Programming Fundamentals

This section introduces the mechanics involved in getting your *API*-based application up and running in a Visual Basic environment.

The API In Action

This section uses step-by-step examples to illustrate common real-world *API* programming tasks. Each example is broken down and discussed in detail.

Plug-Ins

Plug-Ins are a special *API* feature enabling custom UI extensions to the ***Raiser's Edge*** shell. This section explains the concept of plug-ins and how they can be built using *API*.

The Raiser's Edge Object Metaviewer

You can use ***The Raiser's Edge*** Object Metaviewer to use MetaView to examine the type library when you select an object from the outline. This displays the object, its child collections, and shows run-time information (such as MetaField).

API Code Samples

API code samples can be found in the RE7\Help\Samples\API directory on each workstation. Review this section for a brief description of the available code samples.

Plug-In Code Samples

Plug-In code samples can be found in the RE7\Help\Samples\Plugins directory on each workstation. Review this section for a brief description of the available code samples.

What is API?

API stands for *Application Programming Interface*. In short, *API* is a way for you to write custom applications while taking advantage of the wealth of code contained within ***The Raiser's Edge***.

Definition...

API (Application Programming Interface). A developer can use *API* to write a specific request for a computer operating system or a software program.

API follows the guidelines of Microsoft's *Component Object Model* (*COM*). Therefore, you may use it from any COM-enabled programming environment, including Microsoft's *Visual Basic*, *Visual C++*, and *Visual Basic for Applications*. We wrote most of the examples included in this guide using *VBA*.

What Can I Do with API?

A programmer trained in *Visual Basic* and *Object-Oriented Programming* can use the *API* to create applications that work with ***The Raiser's Edge***. *API* can be used to access your ***Raiser's Edge*** data from almost any application:

- Create custom form letters within Microsoft *Word* that directly access the latest information from your ***Raiser's Edge*** database.
- Generate up-to-the-minute comparative financial information from within Microsoft *Excel*.
- Build custom forms that aggregate the fields you use most often.
- Exchange information between ***The Raiser's Edge*** and legacy systems in real time.
- Access current ***Raiser's Edge*** data directly from your own Web site.

There are many different ways that you can use *API*. Generally, any programs that can be used with *Visual Basic COM* objects are able to access *API*. Besides writing stand-alone applications using *Visual Basic*, you can use programs that support *VBA*. For example, instead of a traditional mail merge, you can access constituent information from within Microsoft *Word* to generate thank-you letters.

You can also "extend" ***The Raiser's Edge*** shell by building plug-ins or add-ins that can be made available to your end-users while they are in ***The Raiser's Edge***. Plug-ins written using *API* can be added to **Plug-Ins** in ***The Raiser's Edge***. For example, you can create a plug-in to run several queries, export the data to Microsoft *Excel*, and produce reports (complete with charts and graphs). You can also create a plug-in that includes a data entry form to enable end-users to add information to the database in a custom format.

API can be used to access your data through the Internet by using tools available through Microsoft *Internet Explorer*. You can use the *Windows Scripting Host* provided with the Windows 32-bit operating systems to write scripts (similar to batch files) that use *API* to access ***Raiser's Edge*** data.

API gives you access to your data. Many of the data entry forms and search screens you see while using ***The Raiser's Edge*** are available through *API*. You have access to the database through data objects that can be used to add and edit records. Using *Visual Basic*, or another program that supports access to *Visual Basic COM* objects, you can write custom applications to enhance ***The Raiser's Edge*** for your organization.

For more information...

Visit Blackbaud's Web site at www.blackbaud.com for software customization FAQs, code samples, and other helpful information, such as error explanations. The VBA\API Web site page is one of your primary sources of information for customizing your ***Raiser's Edge*** software. You can also send an email to dssupport@blackbaud.com or call 1-800-468-8996 for support assistance.

API vs. VBA

The most important distinction between *VBA* and *API* is that *VBA* is available only when ***The Raiser's Edge*** is actually up and running.

With *API*, you can write fully functional "standalone" programs that have complete access to ***The Raiser's Edge*** data and services. ***The Raiser's Edge*** does not have to be running. *API* is the appropriate solution if you want to write your own "front-end" to the system, or create a customized system that melds ***The Raiser's Edge*** with another specialized functionality.

Programming with *API* requires you to have your own COM-enabled programming language. None of the niceties inherent to *VBA* are present in *API*. For example, *VBA* includes a complete forms design package. However, this is not the case with *API*. If you want to build a UI using *API*, you must do so on your own.

If your goal is to create an entire ***Raiser's Edge*** application or utility, *API* provides you with the perfect blend of structure and flexibility to accomplish this task. You can also use *API* to gain access from other *VBA*-enabled applications. For example, you can build a custom batch entry form in Microsoft *Excel* that adds records to ***The Raiser's Edge***. In this example, a Microsoft *Excel VBA* macro uses *API* to add records to ***The Raiser's Edge***.

API Programming Fundamentals

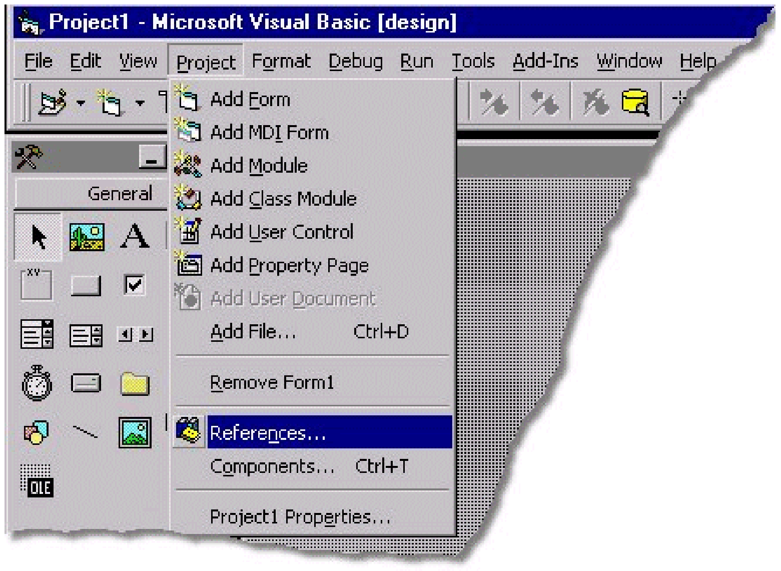
This section introduces the steps involved in using *API* from a *Visual Basic* application.

Using the Type Library from an API Application

When you have the optional module *API* for ***The Raiser's Edge***, you need to set a reference to the type library from any *Visual Basic* project for which you want to gain early-bound access to ***Raiser's Edge*** objects.

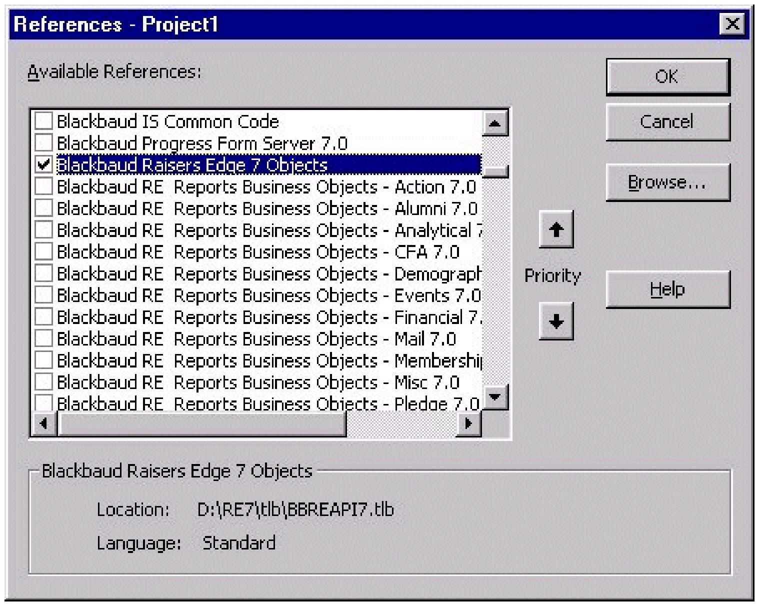
Accessing the References Dialog from Visual Basic 5.0 and Higher

To set a reference to the library from *Visual Basic* 5.0 or higher, create a new *Visual Basic* project and select **Project, References** from the menu bar.



Setting a Reference to The Raiser's Edge Type Library

After you select **References**, the References screen appears. In the **Available References** box, you can see various type library references already set. The most important of these for our purposes is **Blackbaud Raisers Edge 7 Objects**. This is the reference you must set to gain early-bound access to ***Raiser's Edge*** objects.


References - Project1

Available References:

- [ ] Blackbaud IS Common Code
- [ ] Blackbaud Progress Form Server 7.0
- [✓] Blackbaud Raisers Edge 7 Objects
- [ ] Blackbaud RE Reports Business Objects - Action 7.0
- [ ] Blackbaud RE Reports Business Objects - Alumni 7.0
- [ ] Blackbaud RE Reports Business Objects - Analytical 7
- [ ] Blackbaud RE Reports Business Objects - CFA 7.0
- [ ] Blackbaud RE Reports Business Objects - Demograph
- [ ] Blackbaud RE Reports Business Objects - Events 7.0
- [ ] Blackbaud RE Reports Business Objects - Financial 7.
- [ ] Blackbaud RE Reports Business Objects - Mail 7.0
- [ ] Blackbaud RE Reports Business Objects - Membershi
- [ ] Blackbaud RE Reports Business Objects - Misc 7.0
- [ ] Blackbaud RE Reports Business Objects - Pledge 7.0

Blackbaud Raisers Edge 7 Objects

Location: D:\RE7\tlb\BBREAPI7.tlb

Language: Standard

The REAPI Object

The REAPI object (for more information, see the **Programmer's Reference** in the help file) represents the overall program. It provides access to a valid SessionContext needed to initialize any ***Raiser's Edge*** object. It is important you maintain a reference to the *API* object throughout the lifetime of your program; once this object is released, your connection to ***The Raiser's Edge*** is closed.

Use the following syntax to create an *API* object reference from *Visual Basic*:

```
'Create a new REAPI object and set a modular reference to it
Set moREAPI = New REAPI
```

To use this *API* object, you must initialize it. The *API* object has an Init function for this purpose:

REAPI.Init (*sSerialNumber* As String, [*sUserName* As String], [*sPwd* As String], [*DatabaseNumber* As Long], [*sThirdPartyVendor* As String], [lAppMode As AppMode = amStandalone]) As Boolean

This function returns a Boolean indicating the result of the connection attempt.

Part	Description
sSerialNumber	Required. You do not have to enter a serial number, but the parameter is required. You can find your serial number by selecting **Help**, **About the Raiser's Edge** from the menu bar. Use double quotes " ".
sUserName	Optional. A string expression containing a valid user name for the ***Raiser's Edge*** database for which you are attempting to connect. If the **User name** and **Password** fields are blank, the login form appears when the Init method runs.
sPwd	Optional. A string expression containing a valid password for the user name specified above. If both **User name** and **Password** are supplied, the login form does not display.
DatabaseNumber	Optional. A long expression representing the position of the desired database within the login list. Note the standard sample database is always represented by 50. The first live database is usually represented by 1. If the optional parameter is not defined, the database appears to the user, enabling them to select the desired database.
sThirdPartyVendor	Optional. Reserved for third party vendors.
lAppMode	Optional. A long expression indicating whether this application is operating standalone or as a server. If omitted, ***Raiser's Edge*** assumes this is a standalone application.
sSerialNumber	Required. You do not have to enter a serial number, but the parameter is required. You can find your serial number by selecting **Help**, **About the Raiser's Edge** from the menu bar. Use double quotes " ".

Code Sample

To connect to the sample database as "Supervisor" using the default password of "Admin", use the following code:

```
'Initialize the REAPI object and attempt to connect to the RE7 sample database
If Not moREAPI.Init("WRE11111", "Supervisor", "Admin", 10) Then
    MsgBox "Cannot connect to database", vbOKOnly Or vbInformation
    Exit Sub
End If
```

The AppMode Property

AppMode is a read-only property you can use to determine whether the application is running standalone or as part of a server (for example, on a Web server).

```
'Are we running standalone?
If moREAPI.AppMode = amStandalone Then
    'Do standalone code
Else
    'Do server code
End If
```

The GetAvailableRegistryKeys Method

This method returns an array containing the registry key root for each installed ***Raiser's Edge*** database.

```
'Check the registry root of the first installed Raiser's Edge database
Debug.Print moREAPI.GetAvailableRegistryKeys(1)
```

Please remember...

The *API* object does not need to be initialized to access this method.

If you have multiple ***Raiser's Edge*** databases installed with *API* support, you can use this method to present the end-user with a list of available databases.

For example, the following code populates a combo box with a list of available ***Raiser's Edge*** databases:

```
Private Sub UserForm_Initialize()

    Dim lCntr As Long
    Dim vDatabases As Variant

    'Create an instance of The Raiser's Edge 7 API
    Set moREAPI = New REAPI

    'Get a list of available RE7 databases
    vDatabases = moREAPI.GetAvailableRegistryKeys

    'Load a combo with the available choices
    With cboDatabases
        .Clear
        For lCntr = 1 To UBound(vDatabases)
            .AddItem vDatabases(lCntr)
        Next lCntr
    End With

End Sub
```

Please remember....

Once you have access to the Registry Key, you can extract descriptive information from the registry. *to display for your end-user*. The registry key will be returned as \Software\Blackbaud\REINI_## where ## is the Database ID number.

Three useful keys are:

Key	Sample
<*key*>\GENERAL\DB_DESCRIPTION	***The Raiser's Edge*** sample database
<*key*>\GENERAL\DSN	RE7_SAMPLE
<*key*>\GENERAL\SYSTEMPATH	C:\Program Files\RE7

The LastErrorMessage Property

LastErrorMessage is a read-only property you can use to display the reason for an Init method failure.

```
'Initialize the REAPI object and attempt to connect to the RE7 sample database
If Not moREAPI.Init("WRE11111", "Supervisor", "Admin", 10) Then
    MsgBox "Cannot connect to database for the following reason: " _
           moREAPI.LastErrorMessage, vbOKOnly Or vbInformation

    Exit Sub
End If
```

The QueryShutDown Method

Place this method in the QueryUnload event of your main form to make sure all non-modal ***Raiser's Edge*** forms unload. QueryShutDown returns False if one or more of the non-modal forms cannot unload.

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
     'Make sure all non-modal RE forms are closed
    If Not moREAPI.QueryShutDown Then
        Cancel = True
    End If
End Sub
```

The SessionContext Property

The SessionContext holds information regarding the state of the active instance of the ***Raiser's Edge*** application. It is needed to initialize all other objects in the system.

```
'Create a new Gift object and initialize it with our current SessionContext
Set oGift = New CGift
oGift.Init moREAPI.SessionContext
```

The SessionContext is the most popular method used in ***The Raiser's Edge***, although it is not used during the initial log in. Each time a top-level object is created, it must be initialized with a valid SessionContext.

The SignOutOnTerminate Property

If set to True (-1), this property forces your instance of ***The Raiser's Edge*** to log off when the REAPI object is released.

```
'This will cause the application to "sign out" of The Raiser's Edge
moREAPI.SignOutOnTerminate = True
Set moREAPI = Nothing
```

REServices Object

The REServices object wraps up a number of common forms, objects, and collections and provides you with access to them through the CreateServiceObjects method. The objects available are enumerated in bbServiceObjects (for more information, see the **Programmer's Reference** in the help file).

The CreateServiceObject Method

REServices exposes a number of commonly used forms, objects, and collections through the CreateServiceObject method. This example uses CreateServiceObject() to create an instance of the IBBReportInstances collection object, providing you with access to all the constituent profiles that have been created.

```
'Create a reference to an IBBReportInstances collection
    Dim oReports As IBBReportInstances

     'Create an instance of the IBBReportInstances collection
    Set oReports = REServices.CreateServiceObject(bbsoReportInstances)

    'Initialize the collection with the SessionContext and the
     '    report type for Constituent Profiles
    oReports.Init goSessionContext, ReR_Constituent
```

Refer to bbServiceObjects (for more information, see the **Programmer's Reference** in the help file) for a list of Service Objects that can be created.

Class Names

Three methods are provided as part of the REServices object to return the full class name of the creatable objects that are part of the *API*. The full class name uses the syntax *appname***.***objecttype* (for example, *ConstitData7.CRecord*). The three methods include:

- *GetProgIDForDataObject()* accepts a member of the bbDataObjConstants (for more information, see the **Programmer's Reference** in the help file) enum as a parameter and returns the full class name for a data object.
- *GetProgIDForUIObject()* accepts a member of the bbDataObjConstants (for more information, see the **Programmer's Reference** in the help file) enum as a parameter and returns the full class name for a user interface object.
- *GetProgIDForMetaProvider()* accepts a member of the bbMetaObjects (for more information, see the **Programmer's Reference** in the help file) enum as a parameter and returns the full class name for a meta object.

Creating Objects by Application Name and Class ID

With the class name, you can use the *CreateObject()* method that is part of *Visual Basic* to create an instance of the class.

```
Dim oRecord As Object
   Dim sClassName As String

    'Get the class name for the Constituent data object
   sClassName = REServices.GetProgIDForDataObject(bbdataConstituent)

    'Create the data object by the Class Name
   Set oRecord = CreateObject(sClassName)
```

The API In Action

In other sections, you learned about *API* and how to start using it. This section uses examples to demonstrate various *API* programming scenarios. After reviewing these code samples, you should be ready to build your own custom applications using *API*.

We have created sample *Visual Basic* programs for each of the following examples. In many cases, the sample code contains additional information and examples that may be helpful. All these samples can be found in RE7\Help\Samples directory on each workstation.

Accessing The Raiser’s Edge API

This example takes you through the first step in using *API* and connecting to ***The Raiser’s Edge*** through the REAPI Object. The sample program demonstrates three different approaches, each with its own advantages and disadvantages.

You first need to create a global object reference to an REAPI object in the declaration section of your project. You want the reference to be global since it creates and initializes only once when you start your program. Use the object in various places throughout your program to manipulate other ***Raiser’s Edge*** objects, then destroy the object when your program terminates.

```
'Place this in the declarations section of your program
   Public goREAPI as REAPI
```

```
 'Place this in the startup section of your program
   Set goREAPI = New REAPI
'This will force an 'Exit and Sign Out' when your goREAPI object is destroyed
   goREAPI.SignOutOnTerminate = True
```

```
'Place this in the close down section of your program
   Set goREAPI = Nothing
```

REAPI.Init

The REAPI.Init() method is called just after startup. It requires you to provide up to four pieces of information to establish a connection to the database, depending on the method you use. The information you need includes:

Database Serial Number. This is a unique number assigned to your database. If you have multiple databases installed, each database has its own serial number. For example, if you install the Sample Database, it is assigned WRE11111.

User Name. User names are set up under the **Security** link in *Admin*.

Password. When a user is created, he is assigned a password.

Database Number. If you have multiple databases installed, each has a number assigned to it. For example, if you install the Sample Database, it is assigned the number 50.

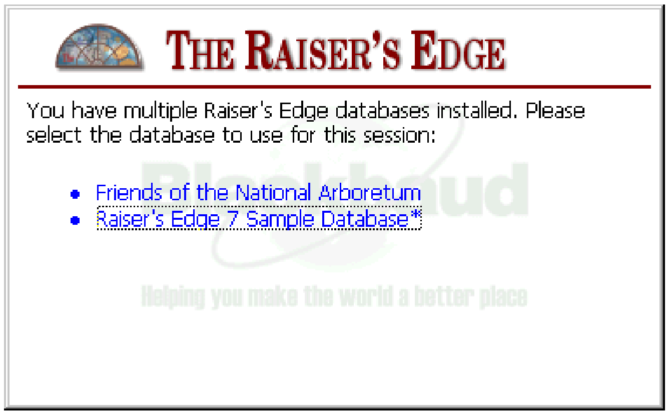
You must always supply the database serial number. The other three parameters are optional when calling the REAPI.Init() method.

Using The Raiser's Edge Login Form

This is the easiest method you can use to connect to ***The Raiser's Edge***.

```
bLoginOK = goREAPI.Init(SERIAL_NUMBER)
```

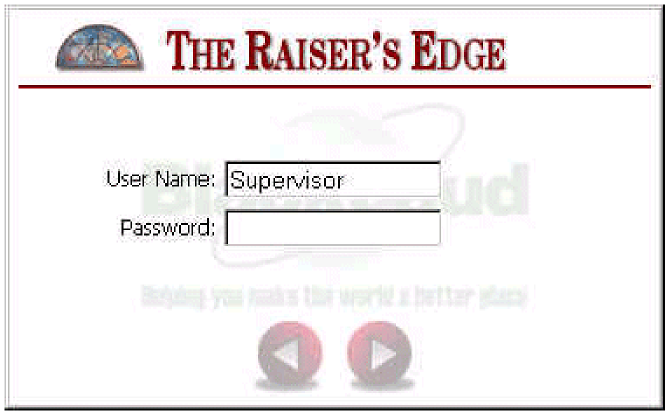
In this case, you specify only the database serial number. Because you did not specify a database number, the initialization code determines if you have multiple databases installed. If that is the case, the following screen appears so end-users can select the database they want to enter.



You have multiple Raiser's Edge databases installed. Please select the database to use for this session:

- Friends of the National Arboretum
- Raiser's Edge 7 Sample Database*

After selecting a database, or if you have only one database installed, the program prompts for a user name and password. If you specify a user name in addition to the database serial number, the user name defaults to the **User Name** field on the login screen.



User Name: Supervisor
Password:

If you specify both user name and password, the login screen does not appear.

Bypassing the Login Form

If you have all the required information, you can connect directly to a ***Raiser's Edge*** database and bypass the login forms. In this case, assuming all information is valid, you connect directly to the database specified and are not prompted.

```
bLoginOK = goREAPI.Init(SERIAL_NUMBER, "Supervisor", "Admin", 10)
```

This method is not ideal because it makes the user password accessible to any end-user with access to your program code. However, it may be useful in cases where you need an application to connect to the database without end-user input. For example, you may want an application to extract information from the database during off-hours and you need this application to be launched by the last person out each night. Using this method, you can permit anyone to run the application with Supervisor rights without distributing your password.

A Custom Login Form

This option is essentially the same as Option 2, but you create a UI for your end-users to select a database and provide a user name and password.

```
bLoginOK = goREAPI.Init(SERIAL_NUMBER, sUserName, sPassword, lDatabaseNumber)
```

The REAPI object provides the method, .GetAvailableRegistryKeys(), which returns a list of Registry Keys associated with each ***Raiser's Edge*** database installed. With this information, you can extract additional information from the registry, such as the database description, the DSN, and the System Directory (the location of ***The Raiser's Edge*** program files).

Addressees and Salutations

The bbosoAddrSalProvider Service Object provides you with access to the list of addressees and salutations in your program. Through this object, you can build salutations based on a constituent's record, or by selecting a specific salutation ID and applying that salutation format.

```
'Place this in the declarations section of your program
    Public goREAPI As REAPI
    Public goREServices As REServices
```

```
'Place this in the startup section of your program
    Set goREAPI = New REAPI

    Set goREServices = New goREServices
    goREServices.Init goREAPI.SessionContext
```

```
'Place this in the close down section of your program
    goREServices.CloseDown
    Set goREServices = Nothing

    Set goREAPI = Nothing
```

Methods are provided to load one or all of the addressees and salutations for a constituent. *LoadAddrSalCombo()* loads the Addressee/Salutations into a combo box (cboAddrSals). The *GetPrimaryAddrSal()* function returns the Primary Addressee for the record passed in. Both of these sample routines require an initialized instance of a CRecord object.

```
'Populates a combobox with the formatted Addressee/Salutations for a
'  selected constituent
Private Sub LoadAddrSalCombo(oRecord as CRecord)

    Dim oAddrSal As IBBAddrSalProvider
    Set oAddrSal = goREServices.CreateServiceObject(bbsoAddrSalProvider)

    With oAddrSal
        .Init goREAPI.SessionContext
        .LoadComboWithAddrSals cboAddrSals, oRecord, False
        .CloseDown
    End With

    oAddrSal.CloseDown
    Set oAddrSal = Nothing

End Sub
```

```
'Returns the formatted Primary Addressee for a selected constituent
Private Function GetPrimaryAddrSal(oRecord as CRecord) As String

    Dim oAddrSal As IBBAddrSalProvider
    Set oAddrSal = goREServices.CreateServiceObject(bbsoAddrSalProvider)

    With oAddrSal
        .Init goREAPI.SessionContext

        'To build the Addr/Sal from the record, set lAddrSalToUse (3rd parameter)
         '  and lAddrSalToOtherwise (4th parameter) to either
         '   -1 for the Primary Addressee
         '   -2 for the Primary Salutation
         '   -3 for the Key Name
        GetPrimaryAddrSal = .BuildAddrSal(oRecord, BN_USEADDRSALFROMRECORD, _
                                          -1, 0, 0, 0)
        .CloseDown
    End With

    oAddrSal.CloseDown
    Set oAddrSal = Nothing

End Function
```

The Annotation Form

The Annotation Service Object requires a parent record that has been initialized and loaded. When the form displays, a reference to the parent object is passed in. The ShowAnnotationForm() routine displays the Annotation form so you can add, edit or delete the annotation from a constituent's record.

```
'Place this in the declarations section of your program
    Public goREAPI As REAPI
    Public goREServices As REServices
```

```
'Place this in the startup section of your program
    Set goREAPI = New REAPI

    Set goREServices = New goREServices
    goREServices.Init goREAPI.SessionContext
```

```
'Place this in the close down section of your program
    goREServices.CloseDown
    Set goREServices = Nothing

    Set goREAPI = Nothing
```

To display the Annotate form, create and load a CRecord object and pass it to ShowAnnotationForm():

```
Private Sub ShowAnnotationForm(oRecord As CRecord)
    Dim oAnnotationFrm As CAnnotationForm
    Set oAnnotationFrm = goREServices.CreateServiceObject(bbsoAnnotationForm)

     'Load the annotation form with the selected data object
    With oAnnotationFrm
        .Init goREAPI.SessionContext
        .ShowAnnotationForm oRecord, Me
        .CloseDown
    End With
    Set oAnnotationFrm = Nothing

     'Save the data object with the new annotation
    With oRecord
        .Save
        .CloseDown
    End With

End Sub
```

Code Tables and the Table Lookup Form

Code tables provide advantages such as standardizing end-user input and increases data entry speed. Two Service Objects enable you to access and manipulate code tables to gain these advantages in your application. *bbsoTableLookupServer* provides access to code tables (for more information, see the **Programmer's Reference** in the help file) through the standard code table lookup form. From this form, end-users can add, edit, delete, or select code table entries. *bbsoCodeTableServer* gives you access to the code table entries for a specific code table and also provides access to static code tables (for more information, see the **Programmer's Reference** in the help file). You can also load table entries directly into combo boxes, or you can retrieve a variant array containing the table entry descriptions and their numeric IDs.

```
'Place this in the declarations section of your program
    Public goREAPI As REAPI
    Public goREServices As REServices
```

```
'Place this in the startup section of your program
    Set goREAPI = New REAPI

    Set goREServices = New goREServices
    goREServices.Init goREAPI.SessionContext
```

```
'Place this in the close down section of your program
    goREServices.CloseDown
    Set goREServices = Nothing

    Set goREAPI = Nothing
```

bbsoCodeTableServer

LoadCodeTableCombo() loads a combo called *cboCodeTable* with the Address Types table entries. *LoadCodeTableArray()* returns a two dimensional variant array. The first dimension has a lower bound of 0 and an upper bound of 1. The second dimension has a lower bound of 1 and an upper bound equal to the number of table entries.

```
Private Sub LoadCodeTableCombo()

    Dim oCodeTableServer As CCodeTablesServer
    Set oCodeTableServer = goREServices.CreateServiceObject(bbsoCodeTablesServer)

    cboCodeTable.Clear

     'This will load a combobox with Address Types
    With oCodeTableServer
        .Init goREAPI.SessionContext
        .LoadCombo cboCodeTable, tbnumAddressTypes, False
        .CloseDown
    End With

    Set oCodeTableServer = Nothing

End Sub
```

```
Private Function LoadCodeTableArray() As Variant

    Dim oCodeTableServer As CCodeTablesServer
    Set oCodeTableServer = goREServices.CreateServiceObject(bbsoCodeTablesServer)

     'This will load an Array with Address Type descriptions and their ID
    With oCodeTableServer

        .Init goREAPI.SessionContext

        '.CodeTableGetDataArray: Returns a two dimensional variant array containing
         '  the ID in vAry(0, n) and the description in vAry(1, n)

        Dim vAry As Variant
        Dim l As Long

        vAry = .CodeTableGetDataArray(tbnumAddressTypes)


        oCodeTableServer.CloseDown

    End With

    Set oCodeTableServer = Nothing

    LoadAddressTypeCodeTableArray = vAry

End Function
```

bbsoTableLookupServer

ShowCodeTableForm() displays the standard table entry form and places the return value into the label *lblLookup*.

```
Private Sub ShowCodeTableForm()

    Dim oTableLookupHandler As CTableLookupHandler

    Set oTableLookupHandler = REServices.CreateServiceObject(bbsoTableLookupServer)

    With oTableLookupHandler
        .Init REAPI.SessionContext
        .ShowForm tbnumAddressTypes, oFormToCenterOn:=Me

        If .Canceled Then
            lblLookup.Caption = ""
        Else
            lblLookup.Caption = "TableEntries.ID: " & .SelectedItem
        End If

        .CloseDown
    End With
    Set oTableLookupHandler = Nothing

End Sub
```

Listing Records

To create any ***Raiser's Edge*** object, you need a valid SessionContext (for more information, see the **Programmer's Reference** in the help file). The SessionContext object contains all the end-user's connection information and can be accessed from the REAPI property, *SessionContext*. These code samples assumes the global variable *goREAPI* has been declared and initialized.

This code sample moves through the CRecords collection and adds constituent names to a List box. The second code sample below demonstrates how to move through collections of top level objects.

```
'Create a reference to a CRecord and a CRecords object
    Dim oRecord As CRecord
    Dim oRecords As CRecords

'Create an instance of the CRecords object
' Using the SessionContext from the REAPI,
' Set the lFilter, so we only pull back constituents,
' Set the bReadOnly parameter so we don't lock up the records
    Set oRecords = New CRecords
    oRecords.Init goREAPI.SessionContext, lFilter:=tvf_record_Constituents, _
                    bReadOnly:=True

'Loop through the oRecords collection and pull out the names
    For Each oRecord In oRecords
        With lstRecords
            .AddItem oRecord.Fields(RECORDS_fld_FULL_NAME)
        End With
        'Only pull the first 10 records for this demonstration
        If lstRecords.ListCount > 10 Then Exit For
    Next

'Clean up the object references
    oRecord.CloseDown
    Set oRecord = Nothing

    oRecords.CloseDown
    Set oRecords = Nothing
```

Media and Notepads

Media and Notepad records can be edited through the Media and Notepads Service Objects. Both have similar properties, methods, and require a parent record that has been initialized and loaded. Before the form displays, a reference on the form is set to the parent's Media or Notepad collection. The *ShowMedia()* routine displays the media form so you can add, edit, or delete media objects from a constituent's media collection.

```
'Place this in the declarations section of your program
    Public goREAPI As REAPI
    Public goREServices As REServices
```

```
'Place this in the startup section of your program
    Set goREAPI = New REAPI

    Set goREServices = New goREServices
    goREServices.Init goREAPI.SessionContext
```

```
'Place this in the close down section of your program
    goREServices.CloseDown
    Set goREServices = Nothing

    Set goREAPI = Nothing
```

To display the Media form, create and load a CRecord object and pass it to ShowMediaForm(). The end-user can then modify the media objects from the constituent's Media collection.

```
Private Sub ShowMediaForm(oRecord As CRecord)

    'Create an instance of the CMediaForm object and create
     '    the service object bbsoMediaForm
    Dim oMediaFrm As CMediaForm
    Set oMediaFrm = goREServices.CreateServiceObject(bbsoMediaForm)

     'Load the Media form with the selected data object
    With oMediaFrm
        .Init goREAPI.SessionContext

        'Set a reference in oMediaFrm to the collection of media
         '    objects on the parent
        Set .MediaObjects = oRecord.Media

        .ShowForm Me
        .CloseDown
    End With
    Set oMediaFrm = Nothing

     'Save the data object with the new media
    With oRecord
        .Save
        .CloseDown
    End With

End Sub
```

Printing Reports

This example uses the REServices (for more information, see the **Programmer's Reference** in the help file) object. REServices permit access to standard ***Raiser's Edge*** forms, search screen, data objects, and reports (which this example uses). To create an instance of the REServices object, use the following syntax.

```
'Place this in the declarations section of your program
    Public goREServices As REServices
```

```
'Place this in the startup section of your program
    Set goREServices = New goREServices
    goREServices.Init REAPI.SessionContext
```

```
'Place this in the close down section of your program
    goREServices.CloseDown
    Set goREServices = Nothing
```

Once the goREServices object is properly initialized, you can use the CreateServiceObject method to create an instance of the IBBReportInstances object. IBBReportInstances is a collection of IBBReportInstance objects. For more information on the CreateServiceObject method and the IBBReportInstances object, see the **Programmer's Reference** in the help file.

This sample code previews the constituent profiles created by the end-user "Supervisor".

```
'Create a reference to a IBBReportInstance object
    and a IBBReportInstances collection
    Dim oReport As IBBReportInstance
    Dim oReports As IBBReportInstances

'Create an instance of the IBBReportInstances collection
    Set oReports = REServices.CreateServiceObject(bbsoReportInstances)

    'Initialize the collection with the SessionContext and the report
        type for Constituent Profiles
    oReports.Init REAPI.SessionContext, ReR_Constituent

     'Cycle through each report and preview them
    For Each oReport In oReports

        With oReport
            .Init REAPI.SessionContext

             'Process only the Profiles generated by the Supervisor
            If .Property(ReR_Property_CreatedByName) = "Supervisor" Then
                .Process ReR_ProcessOption_Preview
            End If

        End With

        oReport.CloseDown
        Set oReport = Nothing

    Next

     'Clean up the object references
    Set oReports = Nothing
```

Using The Raiser’s Edge Search Screen

This example makes use of the REServices (for more information, see the **Programmer’s Reference** in the help file) object. REServices permits access to standard ***Raiser’s Edge*** forms, reports, data objects, and the search screen (which this example uses). To create an instance of the REServices object, use the following syntax.

```
'Place this in the declarations section of your program
    Public goREServices As REServices
```

```
'Place this in the startup section of your program
    Set goREServices = New goREServices
    goREServices.Init REAPI.SessionContext
```

```
'Place this in the close down section of your program
    goREServices.CloseDown
    Set goREServices = Nothing
```

Once the goREServices object is properly initialized, you can use the CreateServiceObject method to create an instance of the IBBSearchSCreen object. For more information about the Create ServiceObject method and the IBBSearchScreen object, see the **Programmer’s Reference** in the help file.

This code displays an Open screen for constituent records and then loads the selected constituent's name into a list box.

```
'Create an instance of an IBBSearchScreen object and create
    the service object bbsoSearchScreen
    Dim oSearch As IBBSearchScreen
    Set oSearch = REServices.CreateServiceObject(bbsoSearchScreen)

'Create a CRecord object to hold the returned constituent
    Dim oRecord As CRecord

    With oSearch
        .Init REAPI.SessionContext

        .AddSearchType SEARCH_CONSTITUENT 'Look only for constituents

        .ShowSearchForm

        If .SelectedID > 0 Then
            Set oRecord = .SelectedDataObject

            lstRecords.AddItem oRecord.Fields(RECORDS_fld_FULL_NAME)

            oRecord.CloseDown
            Set oRecord = Nothing
        End If
    End With

'Clean up the object references
    oSearch.CloseDown
    Set oSearch = Nothing
```

Gift Batch

A wrapper class exists for you to access the *API*. The wrapper, CBatchAPI, along with a limited number of other classes, exposes the Batch DLLs that provide a way for you to programmatically drive gift batch. To drive gift batch through ***The Raiser's Edge*** *API*, use the following code samples.

Create a New Batch

```
Private Sub CreateBatch()

    Dim oBatchAPI As CBatchAPI
    Dim oBatchFields As CBatchFields

    Set oBatchAPI = New CBatchAPI
    With oBatchAPI
        .Init REAPI.SessionContext

        'Add batch fields
        Set oBatchFields = .BatchFields
        With oBatchFields
            SetupBatchField .Add(), GIFT_fld_Constit_ID
            SetupBatchField .Add(), GIFT_fld_Amount
            SetupBatchField .Add(), GIFT_fld_Fund
            SetupBatchField .Add(), GIFT_fld_Date
            SetupBatchField .Add(), GIFT_fld_Post_Date
            SetupBatchField .Add(), GIFT_fld_Post_Status
        End With
        Set oBatchFields = Nothing

        .Save
        .CloseDown
    End With
    Set oBatchAPI = Nothing

End Function

Private Sub SetupBatchField(ByVal oBatchField As CBatchField, ByVal lGiftField As
EGiftFields)

    With oBatchField
        .Fields(BatchField_fld_MetaObjectId) = bbmoGIFT
        .Fields(BatchField_fld_FieldNumber) = lGiftField
    End With

End Sub
```

Add Gifts to a Batch

```
Private Sub AddGiftData(ByVal lBatchID As Long)

    Dim oBatchAPI As CBatchAPI
    Dim oTempRecords As CTempRecords

    Dim lCntr As Long

    Set oBatchAPI = New CBatchAPI
    With oBatchAPI
        .Init REAPI.SessionContext
        .Load lBatchID
        Set oTempRecords = .TempRecords

'Add gifts with differing amounts as an example
        For lCntr = 1 To 10
            AddSingleGift oTempRecords.Add(), CCur(CStr(lCntr))
        Next lCntr

        oTempRecords.Save
        Set oTempRecords = Nothing

        .CloseDown
    End With
    Set oBatchAPI = Nothing

End Sub

Private Sub AddSingleGift(ByVal oTempRecord As CTempRecord, ByVal curAmount As Currency)

    Dim oGift As CGift
    Set oGift = New CGift
    With oGift
        .Init REAPI.SessionContext

        If moRecordFinder.Search("Mark Adamson") Then
            .Fields(GIFT_fld_Constit_ID) = moRecordFinder.RecordId
        End If

        .Fields(GIFT_fld_Amount) = curAmount
```

```
 If UCase$(moServices.GetUserPref(USER_fld_FUNDFORMAT)) = "FUND ID" Then
            .Fields(GIFT_fld_Fund) = "GARDEN"
        Else
            .Fields(GIFT_fld_Fund) = "Botanical Garden Fund"
        End If

        Set oTempRecord.DataObject = oGift

        .CloseDown
    End With
    Set oGift = Nothing

End Sub
```

Commit a Batch

```
Private Sub AddGiftData(ByVal lBatchID As Long)

    Dim oBatchAPI As CBatchAPI

    Set oBatchAPI = New CBatchAPI
    With oBatchAPI
        .Init REAPI.SessionContext
        .Load lBatchID

        'Commit with the default posting options, but show the options form to allow changes
        .Commit Nothing, True, Nothing, True
        .CloseDown
    End With
    Set oBatchAPI = Nothing

End Sub
```

Plug-Ins

Plug-ins are specially built applet extensions to ***The Raiser's Edge***. A plug-in does just that, it "plugs in" to ***The Raiser's Edge*** UI, opening up the door to a wide range of custom functionality. Plug-ins can be as simple as an HTML page or a Microsoft *Office* document, or as complicated as a multi-level ActiveX document or interactive spreadsheet.

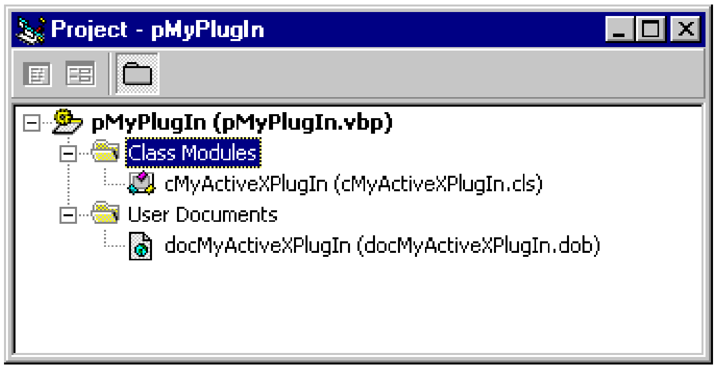
With the flexibility of plug-ins, you can add custom applications and extensions directly into ***The Raiser's Edge***. Plug-ins share the database connection and runtime code resources within the program making them an excellent choice for adding custom functionality without the overhead of having to build a full blown *API* application.

In order to build a plug-in, we must build a special COM dynamic link library (DLL) using *API*. This section details the process.

Creating a Plug-In

The typical plug-in consists of two parts. The first is a class module that implements the IBBPlugIn interface and provides information about the plug-in. The second is a document that provides the UI. The choice of the type of document depends on the application; it can be anything from a simple HTML page to a complex ActiveX Document. The following example uses *Visual Basic* 6.0 to create a COM dynamic link library (DLL) called *pMyPlugIn.DLL*.

Begin by creating a new Active-X DLL project and adding a Class (if a class module is loaded with the default name *Class1*, you can simply rename it) and User Document module.


Project - pMyPlugIn
pMyPlugIn (pMyPlugIn.vbp)
Class Modules
cMyActiveXPlugIn (cMyActiveXPlugIn.cls)
User Documents
docMyActiveXPlugIn (docMyActiveXPlugIn.dob)

A single DLL can contain any number of class modules which implement the IBBPlugIn interface. This enables you to store all your plug-ins in one place, using common forms and code again. For now, a single Class module, *cMyActiveXPlugIn.cls* is the base class for the plug-in. *docMyActiveXPlugIn.dob* is used for the UI.

You also need to set a reference to the RE7 *API* Library (for more information, see “Using the Type Library from an API Application” on page 97). If you use any other object libraries, you need to set these references, as well.

The IBBPlugIn Interface

After adding a reference to the IBBPlugIn interface in the General Declarations section of your plug-in class, you need to fill in each of the plug-in properties and events. The properties provide information to the host, such as the name and description of the plug-in.

Property	Description
DocumentName	Specifies the name of the document to load.
DocumentType	Reserved for future use.
HeaderCaption	Appears in the top frame when the plug-in loads.
HeaderImage	Displays a graphic of your choice to the left of the header caption.
PluginDescription	This is the text displayed in the left **Description** column of the Plug-Ins page.
PluginName	This is the text displayed in the right **Plug-In** column of the Plug-Ins page.

The four events allow you to respond to actions by the system and the end-users.

Event	Description
OnInit	Occurs before all other events, when the host creates an instance of the plug-in object.

OnLoad	Occurs when the plug-in document loads into the shell and before control is passed to the end-user.
OnQueryUnload	Occurs before the document unloads.
OnClosedown	Occurs when the host destroys the document.

The steps below take you through setting up a simple plug-in class.

1. Add the interface reference.

```
Implements IBBPlugIn

'A session context for the application is passed in when the
'  plug-in is initialized (IBBPlugIN_OnInit()).  This may not
'  be necessary, depending on the application
Private moSessionContext As IBBSessionContext

'When the Plug-In is loaded (IBBPlugIn_OnLoad()), the main
'  user interface document is passed in.  This may not be
'  necessary, depending on the application
Private moUserDoc As Object
```

2. Set any initialization information. Note that when a plug-in is first accessed, ***The Raiser's Edge*** host needs to perform several initialization tasks before the plug-in can load. This requires that the *OnInit* and *OnCloseDown* events fire. Therefore, the *OnInit* and *OnClosedown* events cannot be used to determine whether or not the plug-in has been run. You should avoid putting code into these events, with the exception of setting a reference to the SessionContext in *OnInit*, and clearing the reference in *OnCloseDOwn*. *OnLoad* and *OnQueryUnLoad* do not fire during the initialization process and can be used for any required start-up and closedown code.

```
Private Sub IBBPlugIn_OnInit(oREHost As BBInterfaces.IBBShellHost)
    Set moSessionContext = oREHost.SessionContext
End Sub

Private Sub IBBPlugIn_OnLoad(oDoc As Object) 'User Object
    Set moUserDoc = oDoc
End Sub
```

3. Specify the name of the plug-in UI document. The *IBBPlugIn_DocumentName()* property should return the path to the UI file. If you are using user documents for your interface, when you create the DLL, each user document is in the same directory as the DLL with the extension *.vbd.

```
'Uses App.Path to return the path of the RE7 application,
'   and adds the PlugIns path and file name.
Private Property Get IBBPlugIn_DocumentName() As String
    IBBPlugIn_DocumentName = App.Path & "\PlugIns\docMyPlugIn.vbd"
End Property

Private Property Get IBBPlugIn_DocumentType() As BBREAPI7.REShellDocumentTypes
    'Specify the type of document (HTML or ACTIVEX)
     '  - this property is reserved for future use and is not currently used
    IBBPlugIn_DocumentType = redocActiveXDocument
End Property
```

4. Create a user-friendly description for your plug-in. The text displays on the Plug-Ins page in the ***The Raiser's Edge***.

```
Private Function IBBPlugIn_PluginName() As String
    IBBPlugIn_PluginName = "My Plug-In"
End Function

Private Function IBBPlugIn_PluginDescription() As String
    IBBPlugIn_PluginDescription = "Sample RE7 Plug-In"
End Function
```

5. Create a caption for your plug-in and add a graphic, if necessary. This appears at the top of the Plug-Ins page in the ***The Raiser's Edge***. Plug-in header images are 32 x 32 pixels and can be *.jpg, *.gif, or *.bmp format.

```
Private Property Get IBBPlugIn_HeaderCaption() As String
    IBBPlugIn_HeaderCaption = "My Plug-in Header Caption"
End Property

Private Property Get IBBPlugIn_HeaderImage() As String
    IBBPlugIn_HeaderImage = App.Path & "\PlugIns\MyPlugIn.jpg"
End Property
```

6. Close down the plug-in properly. The *IBBPlugIn_OnQueryUnload()* occurs before the plug-in or application closes. Note that linking to a separate HTML page from the shell or switching to another shell menu item causes this event to fire. *OnQUeryUnload* allows you to verify information and cancel the close process if the end-user has not completed all necessary tasks. Setting *bCancel* to true cancels the unload and returns the end-user to the plug-in form. *bShellIsUnloading* is true if the end-user is trying to close ***The Raiser's Edge***.

```
Private Sub IBBPlugIn_OnQueryUnload(bCancel As Boolean, _
                ByVal bShellIsUnloading As Boolean)

    'AllowClose is a public method on the docMyPlugIn user
    '  document.  The routine validates the user input
    '  and determines if the plug-in can be closed.
    If Not moUserDoc.AllowClose Then
        MsgBox "Required field missing."
        bCancel = True
    End If

End Sub
```

```
Private Sub IBBPlugIn_OnClosedown()
    If Not moUserDoc Is Nothing Then
        Set moUserDoc = Nothing
    End If

    If Not moSessionContext Is Nothing Then
        Set moContext = Nothing
    End If
End Sub
```

The User Interface (UI)

You can specify a wide array of document types in the *DocumentName()* property and let ***The Raiser's Edge*** serve as host. For example, to host a Microsoft *Excel* spreadsheet you can use the following code sample.

```
Private Property Get IBBPlugIn_DocumentName() As String
    IBBPlugIn_DocumentName = App.Path & "\Events.xls"
End Property
```

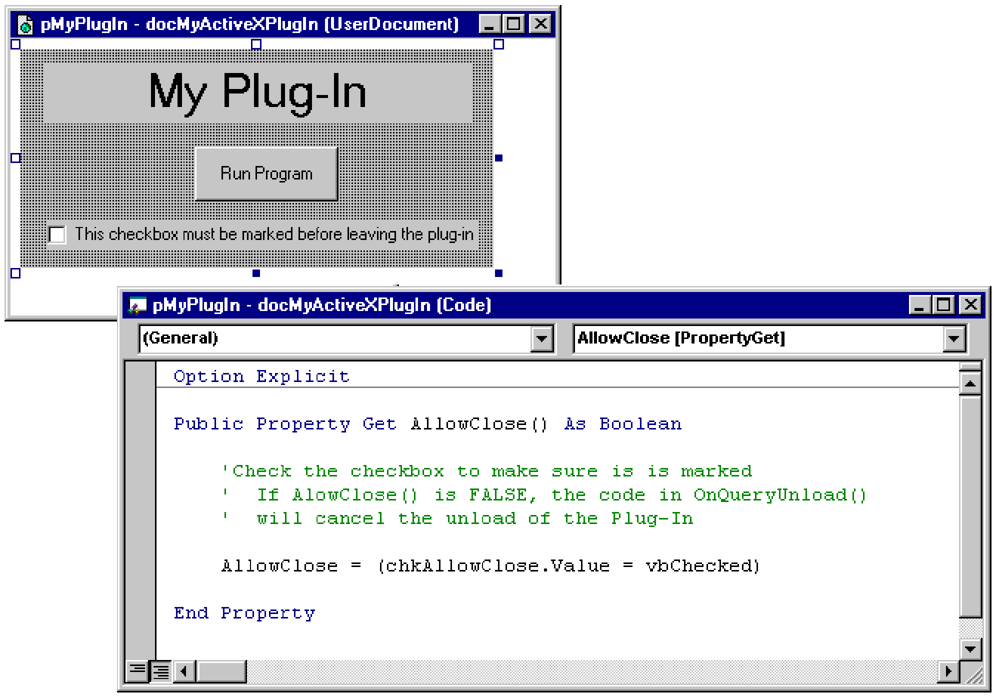
To host a local HTML page, you would use the following code sample.

```
Private Property Get IBBPlugIn_DocumentName() As String
    IBBPlugIn_DocumentName = App.Path & "\MyHTMLPlugIn.Html"
End Property
```

To host an HTML page on the Web, you would use the following code sample.

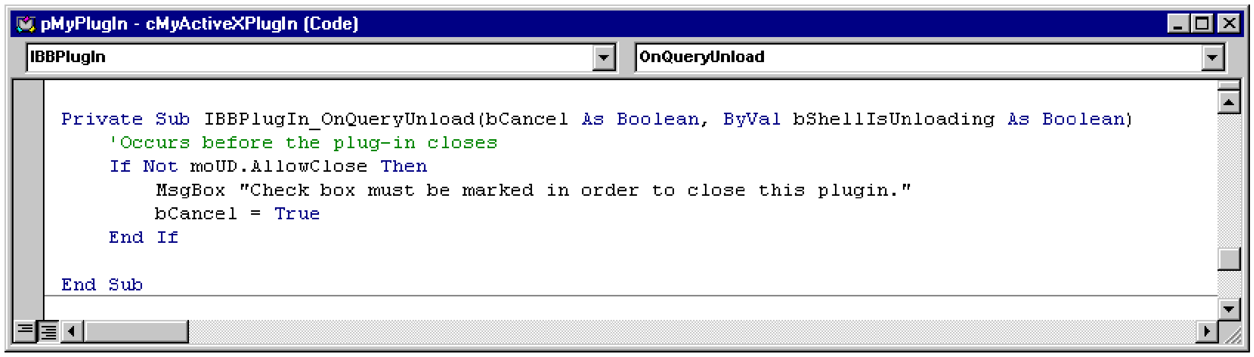
```
Private Property Get IBBPlugIn_DocumentName() As String
    IBBPlugIn_DocumentName = "http://www.blackbaud.com"
End Property
```

To continue the example, we create a user document to provide the UI for this plug-in.


pMyPlugIn - docMyActiveXPlugIn (UserDocument)
My Plug-In
Run Program
This checkbox must be marked before leaving the plug-in
pMyPlugIn - docMyActiveXPlugIn (Code)
(General)
AllowClose [PropertyGet]

```
Option Explicit

Public Property Get AllowClose() As Boolean

    'Check the checkbox to make sure is is marked
    '  If AlowClose() is FALSE, the code in OnQueryUnload()
    '  will cancel the unload of the Plug-In

    AllowClose = (chkAllowClose.Value = vbChecked)

End Property
```

Here, the user document does not perform any specific task. It contains one property, *AllowClose()*, which is used by the *OnQueryUnload()* event of the *cMyActiveXPlugIn.cls*. Unless the checkbox is marked, the end-user cannot close the plug-in. You can use this method to validate end-user input or make sure that all required tasks are completed before closing the plug-in.



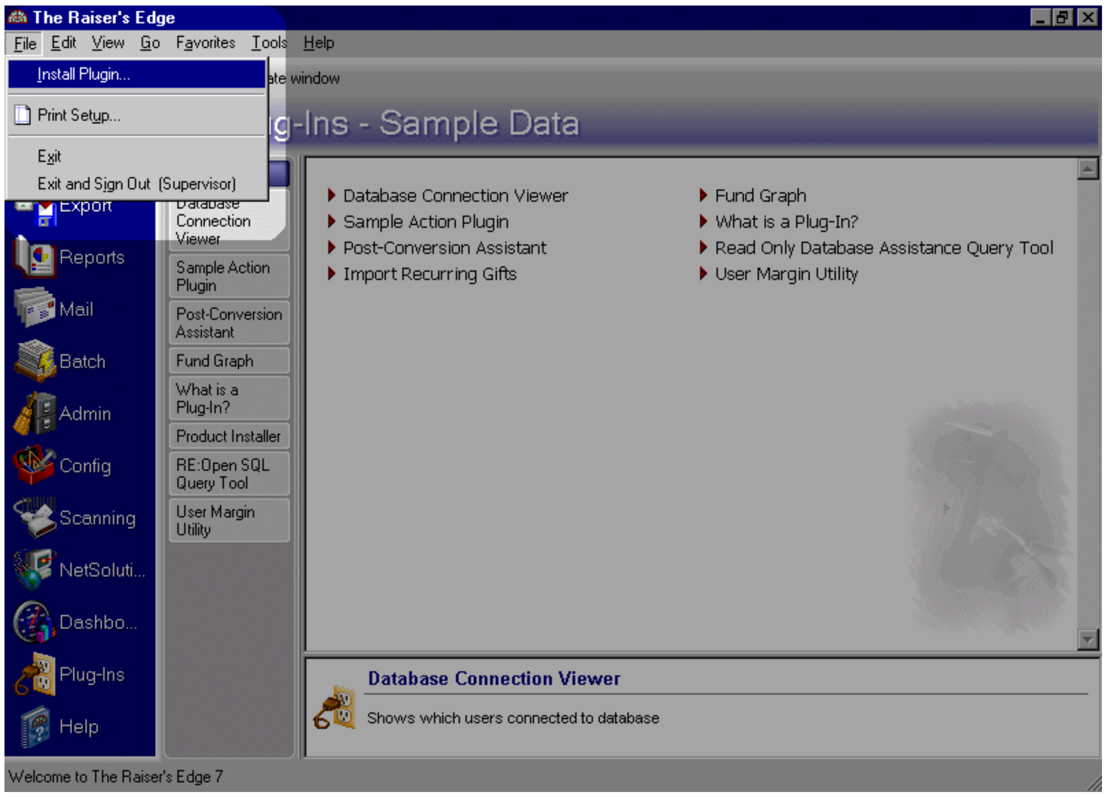
```
Private Sub IBBPlugIn_OnQueryUnload(bCancel As Boolean, ByVal bShellIsUnloading As Boolean)
    'Occurs before the plug-in closes
    If Not moUD.AllowClose Then
        MsgBox "Check box must be marked in order to close this plugin."
        bCancel = True
    End If

End Sub
```

Deploying Your Plug-In

After you compile your plug-in, you need to copy the plug-in DLL and any relevant support files to the RE7\Plugins directory. Once the files are in place, your end-users have access to the plug-in from the **Plug-Ins** link on the Raiser's Edge bar.

Installing the Plug-In

In Plug-Ins, select **File**, **Install Plugin** from the shell menu bar.



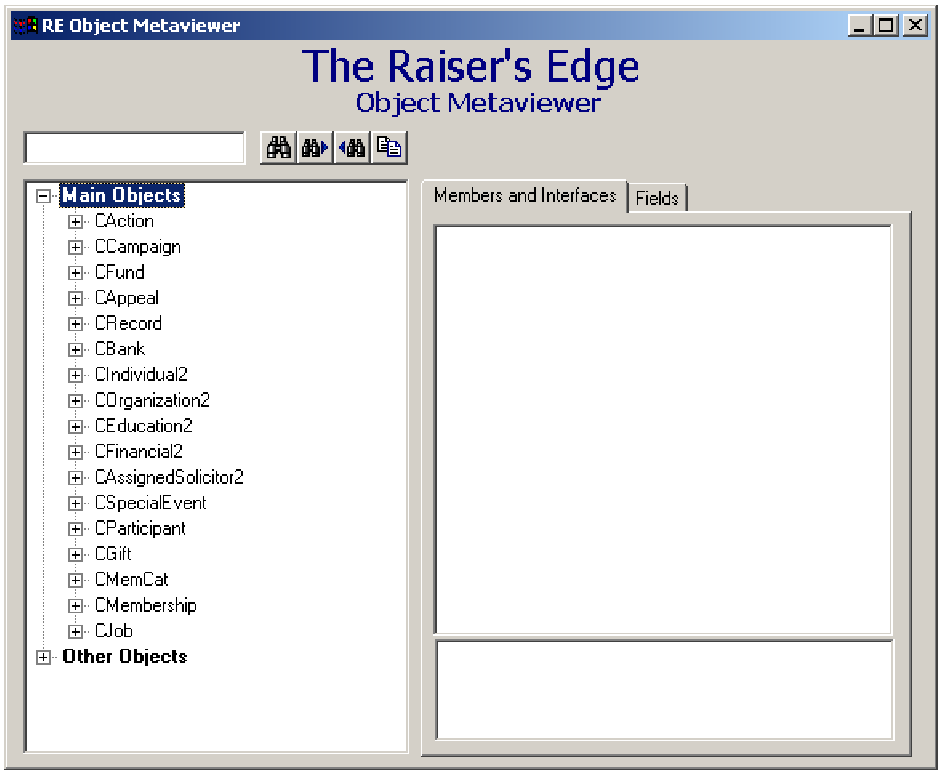
From here, you can select your plug-in DLL and copy the DLL file to the RE7\PlugIns directory.

Make sure all other support documents and files that are referenced by the plug-in are installed on the end-user’s machine. This includes the user document files and the HTML files you are using for the UI.

The Raiser’s Edge Object MetaViewer

Using ***The Raiser’s Edge*** Object Metaviewer, you can use MetaView to examine the type library when you select an object from the outline. This displays the object, its child collections, and shows run-time information (such as MetaField).

To access the Object MetaViewer, double-click REMetaView.exe in your The Raiser’s Edge 7\Help folder on any workstation. The list of nodes under **Main Objects** are ***The Raiser’s Edge*** top level objects.



- Main Objects
  - CAction
  - CCampaign
  - CFund
  - CAppeal
  - CRecord
  - CBank
  - CIndividual2
  - COrganization2
  - CEducation2
  - CFinancial2
  - CAssignedSolicitor2
  - CSpecialEvent
  - CParticipant
  - CGift
  - CMemCat
  - CMembership
  - CJob
- Other Objects

API Code Samples

These samples provide you with basic and advanced concepts of connecting to ***The Raiser’s Edge***. Each sample contains a ReadMe.txt file explaining the steps necessary to use the *API* application. All *API* samples are located in the RE7\Help\Samples\API directory on each workstation.

Sample	Format	Description
Log In	*Visual Basic* 6.0	Demonstrates three methods for connecting to ***The Raiser’s Edge*** database. You can connect directly without end-user interaction, prompt the end-user for login name and password using the standard ***Raiser’s Edge*** login screen, or use a custom login screen.
Search screen	*Visual Basic* 6.0	Uses the standard ***Raiser’s Edge*** search screen to look up records.
List Records	*Visual Basic* 6.0	Using a CRecords object, retrieves a list of Constituent records.
Addressee/ Salutation	*Visual Basic* 6.0	Uses the Addressee/Salutation provider to build addressees and salutations for constituents.
Attribute Types	*Visual Basic* 6.0	A demonstration of the AttributeTypeServer object.

Code Table Server	*Visual Basic* 6.0	This sample shows you how to extract code table information. In addition, it demonstrates the TableLookupHandler object that displays the standard code table form.
Forms	*Visual Basic* 6.0	Uses the CreateServiceObject method of the REServices object to display the search screen, annotation, Media, and Notepad forms.
Misc UI	*Visual Basic* 6.0	Displays several of the miscellaneous UI forms, including Print Setup and the About form. Makes use of the **Quick Find** method of Misc UI interface to find records (an alternative to the standard search screen).
Prog IDs	*Visual Basic* 6.0	Demonstrates an alternate method for creating a data object using the *GetProgIDforDataObject* method of the REServices object.
Reports	*Visual Basic* 6.0	An example of how to automate running reports through *API*.

Plug-In Code Samples

These samples range from a simple outline of a plug-in to more complicated examples that make use of some of the *API* objects. Each sample contains a ReadMe.Txt file explaining the steps necessary to use the plug-in. All Plug-in samples are located in the RE7\Help\Samples\PlugIns directory on each workstation.

For more information...

Visit Blackbaud's Web site at www.blackbaud.com for software customization FAQs, code samples, and other helpful information, such as error explanations. The VBA\API Web site page is one of your primary sources of information for customizing your ***Raiser's Edge*** software. You can also send an email to dssupport@blackbaud.com or call 1-800-468-8996 for support assistance.

Sample	Format	Description
MyPlugIn	*Visual Basic* 6.0	Provides a basic example of the events in user document and HTML document based plug-ins.
TodaysGiving	*Visual Basic* 6.0	Uses the CGifts object to provide a simple summary of giving for the current day. Breaks down the giving into Cash, Pledge, and Other.

Please remember....

We provide programming examples for illustration only, without warranty either expressed or implied, including, but not limited to, the implied warranties of merchantability and/or fitness for a particular purpose. This article assumes that you are familiar with Microsoft *Visual Basic* and the tools used to create and debug procedures. Blackbaud Customer Support can help explain the functionality of a particular procedure but they will not modify, or assist you with modifying, these examples to provided additional functionality. If you are interested in learning more about ***The Raiser's Edge*** optional modules *VBA* and *API*, contact our Sales department at solutions@blackbaud.com.

Index