

# The**Raiser's**Edge<sup>®</sup> Enterprise<sup>™</sup>

VBA Essentials Guide

## **030308**

©2008 Blackbaud, Inc. This publication, or any part thereof, may not be reproduced or transmitted in any form or by any means, electronic, or mechanical, including photocopying, recording, storage in an information retrieval system, or otherwise, without the prior written permission of Blackbaud, Inc.

The information in this manual has been carefully checked and is believed to be accurate. Blackbaud, Inc., assumes no responsibility for any inaccuracies, errors, or omissions in this manual. In no event will Blackbaud, Inc., be liable for direct, indirect, special, incidental, or consequential damages resulting from any defect or omission in this manual, even if advised of the possibility of damages.

In the interest of continuing product development, Blackbaud, Inc., reserves the right to make improvements in this manual and the products it describes at any time, without notice or obligation.

All other products and company names mentioned herein are trademarks of their respective holder.

All other products and company names mentioned herein are trademarks of their respective holder.

RE7Enterprise-VBAEssentialsUK-030308

# VBA Essentials Guide

ESSENTIALS. ....	1
Using This Guide .....	3
Objects and Object Models .....	6
The Raiser's Edge Type Library .....	7
The Raiser's Edge Object Fundamentals .....	11
Data Objects .....	13
Programming Child Objects and Collections .....	21
Adding and Deleting Child Objects .....	30
Filtering Data Object Collections .....	32
Error Handling .....	33
User Interface (UI) Objects .....	34
Raiser's Edge ActiveX Controls .....	37
Service Objects .....	45
Report Objects .....	50
Advanced Concepts and Interfaces .....	80
Custom View: Creating Custom Parts .....	85
 VBA .....	 95
What is VBA? .....	96
API vs. VBA .....	97
The VBA Environment .....	97
Macros .....	101
VBA DLL Macros .....	140
Customising RE:Anywhere for Remote Access Using VBA .....	142
VBA Code Samples .....	147
 INDEX .....	 149

contents

\_\_\_\_\_



## What Is In This Guide?

Using the *VBA Essentials Guide*, your technical staff can learn how to use the optional module *VBA for Advanced Customisation* to customise **The Raiser's Edge** for your organisation's policies and procedures. Using *VBA*, a program developer can create custom business rules, design custom screens, create Web site interfaces, and much more. You can also learn about the following:

- “The Raiser's Edge Object Fundamentals” on page 11
- “Programming Child Objects and Collections” on page 21
- “Advanced Concepts and Interfaces” on page 80
- “Custom View: Creating Custom Parts” on page 85
- “The VBA Environment” on page 97
- “Macros” on page 101
- “VBA Code Samples” on page 147



## 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.co.uk](http://www.blackbaud.co.uk). 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.

W e l c o m e



# Essentials

## Contents

<b>Who is This Guide For?</b> .....	<b>3</b>
Documentation Map .....	4
Programming Language .....	4
Sample Code .....	4
Raiser's Edge Programming Essentials .....	5
<b>Objects and Object Models</b> .....	<b>6</b>
What Are Objects and Object Models? .....	6
The Raiser's Edge Object Model .....	6
Data Objects .....	6
<b>The Raiser's Edge Type Library</b> .....	<b>7</b>
Using Early Bound Objects and the Type Library .....	7
Using the Type Library from VBA .....	8
Accessing the References Dialog .....	9
Setting a Reference to The Raiser's Edge Type Library .....	10
Using the Type Library from an API Application .....	10
Accessing the References Dialog from Visual Basic 5.0 and Higher .....	11
<b>The Raiser's Edge Object Fundamentals</b> .....	<b>11</b>
The SessionContext .....	11
Accessing the SessionContext from VBA .....	12
Accessing the Session Context from API .....	12
Initialising and Releasing Objects .....	13
The Init and CloseDown Methods .....	13
<b>Data Objects</b> .....	<b>13</b>
Data Object Hierarchy .....	14
What Are "Top Level" Objects? .....	14
Loading a Data Object .....	15
How Many Ways Can I Load a Data Object? .....	15
An Alternate Method to Load Data Objects—The Database ID .....	15
Using The Raiser's Edge Search Screen to Load Your Data Object .....	16
Updating Data Objects .....	18
The Fields Property .....	19
Validation and Integrity .....	20
Adding and Deleting Data Objects .....	20
Adding a Record Using a Data Object .....	20
Deleting a Record Using a Data Object .....	21
<b>Programming Child Objects and Collections</b> .....	<b>21</b>
What is a Child Object? .....	21
Child Collection Types .....	22
The Standard Child Collection .....	22
The Child Top Collection .....	26
The Child View Collection .....	28

# Chapter 1



The Top View Collection . . . . .	28
<b>Adding and Deleting Child Objects . . . . .</b>	<b>30</b>
Adding a Child Object . . . . .	30
Deleting a Child Object . . . . .	31
Sorting Collections . . . . .	31
SortField . . . . .	31
SortOrder . . . . .	32
<b>Filtering Data Object Collections . . . . .</b>	<b>32</b>
<b>Error Handling . . . . .</b>	<b>33</b>
Return Code Based . . . . .	33
Error Code Based . . . . .	33
<b>User Interface (UI) Objects . . . . .</b>	<b>34</b>
Data Entry Forms . . . . .	35
Showing a Standard Form . . . . .	35
<b>Raiser's Edge ActiveX Controls . . . . .</b>	<b>37</b>
Data Grid . . . . .	39
Attributes Grid . . . . .	42
Phones/Email/Links Grid . . . . .	45
<b>Service Objects . . . . .</b>	<b>45</b>
Query Objects . . . . .	46
Opening a Query . . . . .	46
Processing a Query Result Set . . . . .	46
Creating Static Queries . . . . .	47
<b>Report Objects . . . . .</b>	<b>50</b>
Reports Categories Collection . . . . .	50
Reports Types Collection . . . . .	52
Report Instances Collection . . . . .	53
Report Objects Sample . . . . .	55
Code Tables Server . . . . .	57
Table Lookup Handler . . . . .	60
Attribute Type Server . . . . .	63
Annotation Form . . . . .	66
Using the Annotation Form Object . . . . .	66
Notepad Form . . . . .	68
Using the Notepad Form Object . . . . .	68
Media Form . . . . .	70
Using the Media Form Object . . . . .	70
Property Viewer . . . . .	73
Using the Property Viewer . . . . .	73
Search Screen . . . . .	75
Using the Search Screen Object . . . . .	75
MiscUI . . . . .	78
Using the MiscUI Object . . . . .	78
<b>Advanced Concepts and Interfaces . . . . .</b>	<b>80</b>
Using the IBBDatObject Interface . . . . .	81
Using the IBBMetaField Interface . . . . .	82
Transactions . . . . .	84
<b>Custom View: Creating Custom Parts . . . . .</b>	<b>85</b>
Custom Parts . . . . .	85
Adding a Custom Part . . . . .	86



This guide provides Visual Basic developers with all the information needed to customise 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.co.uk](mailto:solutions@blackbaud.co.uk).

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.co.uk](http://www.blackbaud.co.uk) for software customisation 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 customising your **Raiser's Edge** software. You can also contact Customer Support for 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 Customisation* 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 organisation's **Raiser's Edge** software package, contact Sales at [solutions@blackbaud.co.uk](mailto:solutions@blackbaud.co.uk). 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.

### Initialising and Releasing Objects

This section outlines the mechanics of creating and destroying objects, as almost every object in *The Raiser's Edge* must be initialised 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 organised. 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

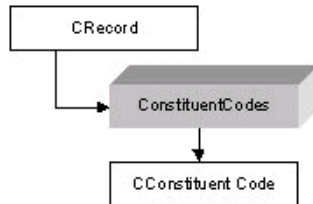
*The Raiser's Edge* object model is best described as a group of object models. Each major record type in the system has its own hierarchical object model. For example, the Gift has a large model comprised of "child" objects such as instalments, split funds, and attributes. The concept of a child object 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 initialise 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
  
```

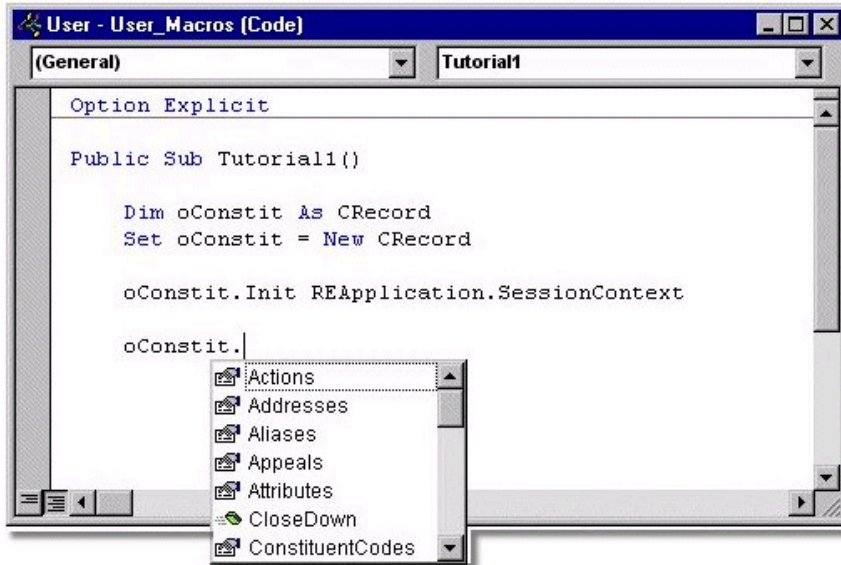
## The Raiser's Edge Type Library

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.



*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
```

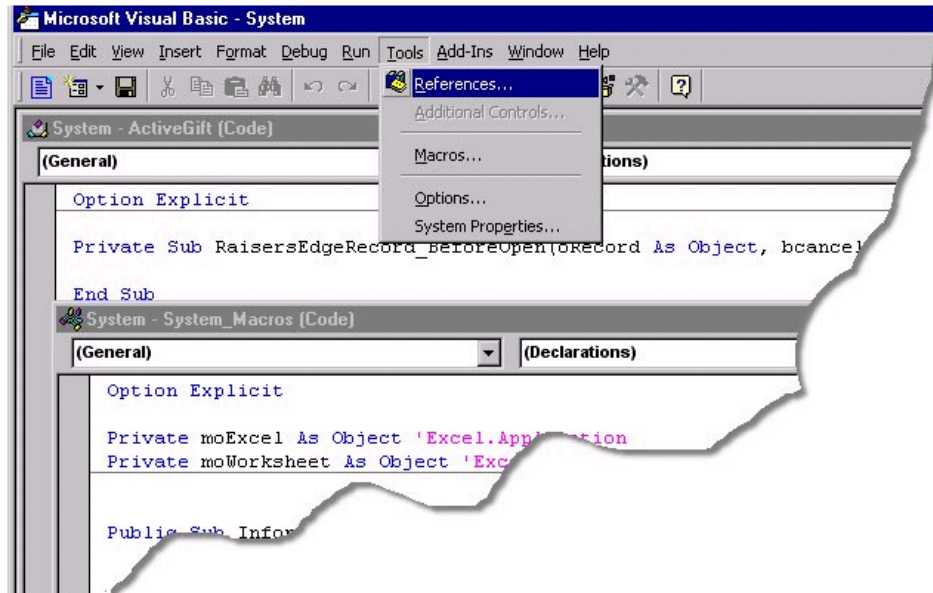
## Using the Type Library from VBA

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

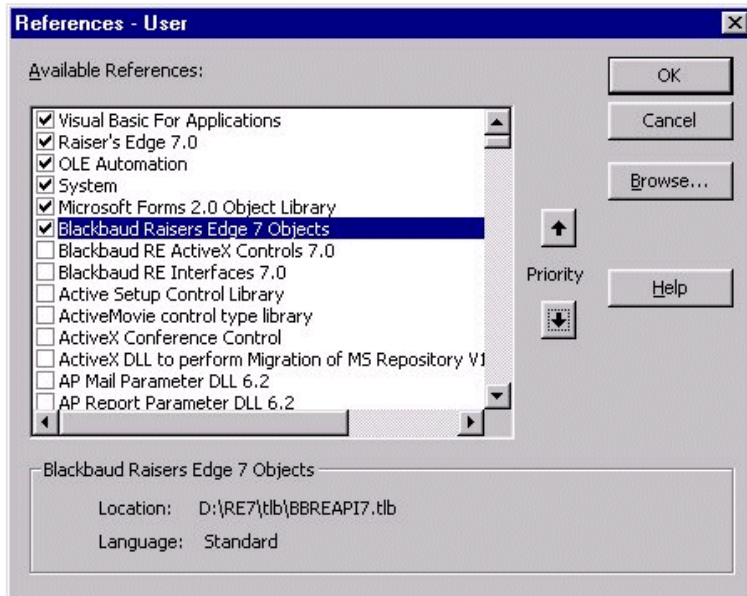
You can manually set a reference to the library by selecting **Tools, References** from the menu bar in VBA.





## 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.



### Note...

If you uncheck 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.

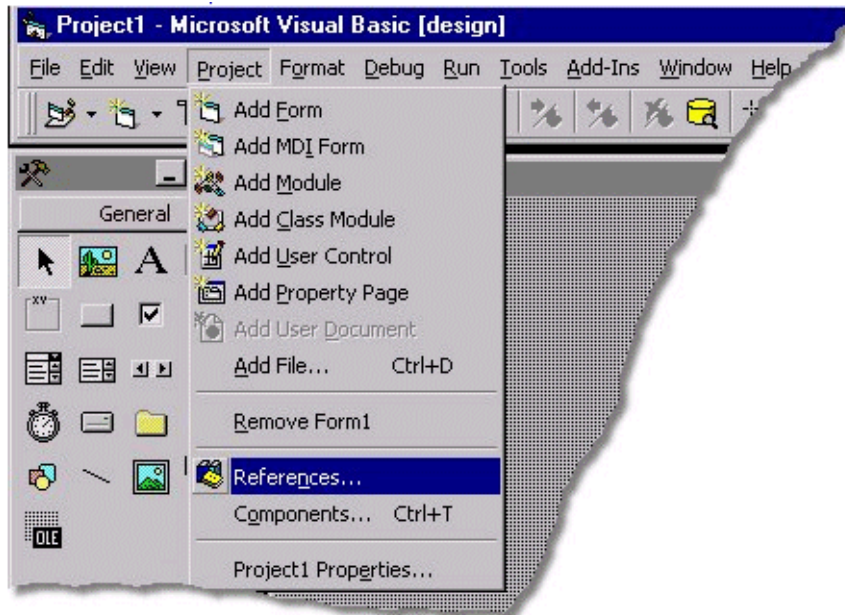
## Using the Type Library from an API Application

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 “Initialising 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 initialised first. All objects are initialised 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 initialise 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 initialised, *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 initialises 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 initialised 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 initialised and available to VBA in the running instance of *The Raiser's Edge*, API must be initialised by the programmer. The code sample accomplishes the same task as the previous VBA sample.

```
Dim oAPI as REAPI

'Initialise 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).

## Initialising and Releasing Objects

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

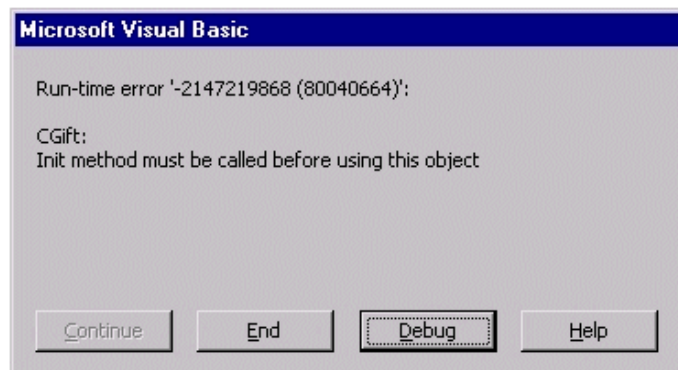
```
Dim oGift as CGift
Set oGift = New CGift

'Initialise 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
```

Initialise (.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 initialising it, a trappable runtime error is raised.



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.

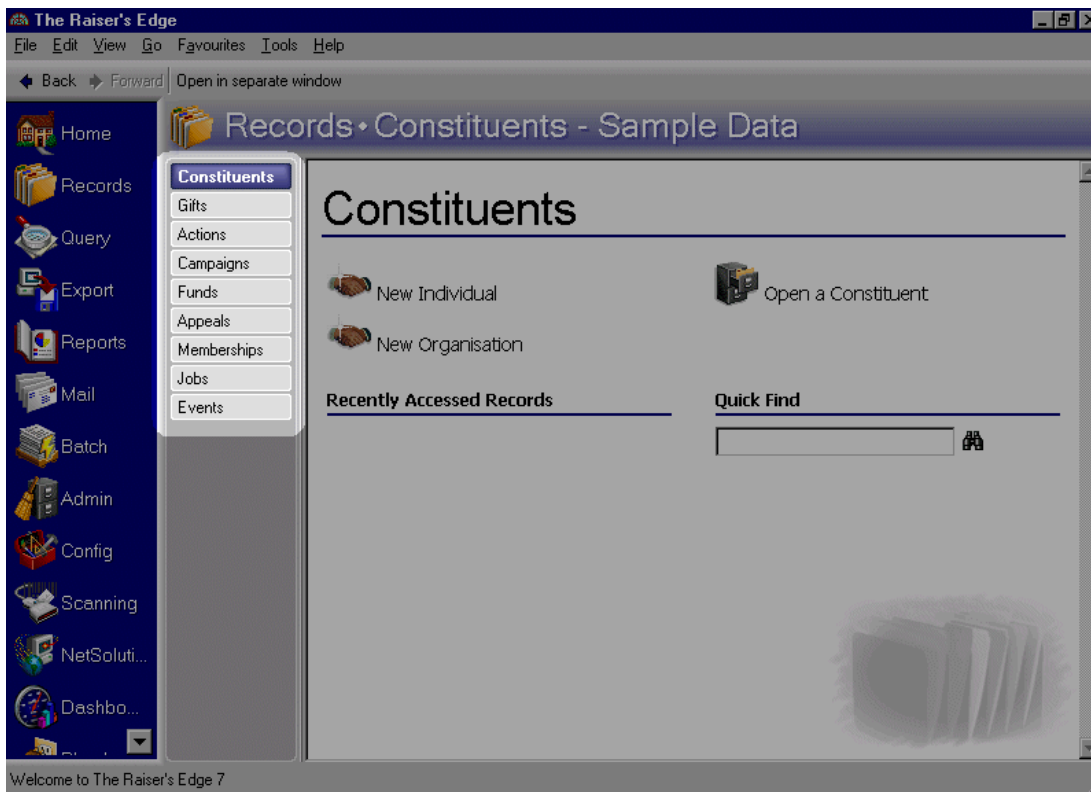
## Data Object Hierarchy

To manipulate a data record in your *Raiser's Edge* system, you initialise 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 familiarise 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.



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.

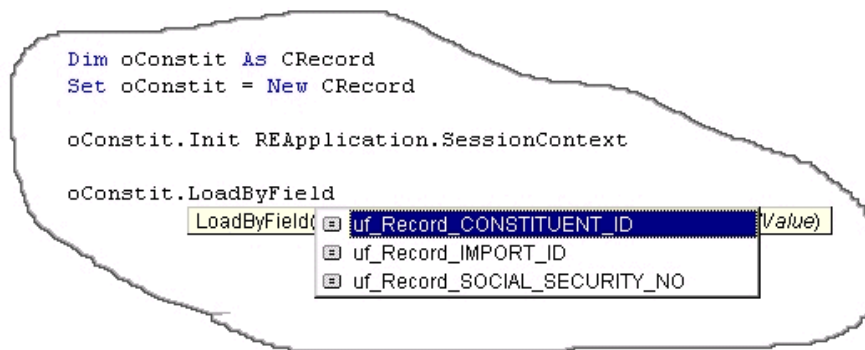
## Loading a Data Object

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.



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 organisation 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 national insurance number of NZ 13 05 77 A, and a database (primary key) of 166.

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

oConstit.Init REApplication.SessionContext

'Load the record via the National Insurance Number
oConstit.LoadByField uf_Record_SOCIAL_SECURITY_NO, "NZ 13 05 77 A"
```

```
'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 customised 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.

Open

Find: Constituent Search using query: <Default>

Open Cancel

Add New Options

Find Now New Search Previous Search

Expand Results

Find Constituents that meet these criteria:

Sur/Org name:	Address lines:
First Name:	City:
Constituent ID:	County:
NIN:	Postcode:
Membership ID:	Class of:
Bank acct. no.:	Credit card no.:

☒ Display inactive constituents ☐ Check spouse name ☐ Exact match only ☐ Check nickname  
☐ Display deceased constituents ☐ Check aliases ☐ Check contact name

## Updating Data Objects

Previously in this guide, we explored “Initialising 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

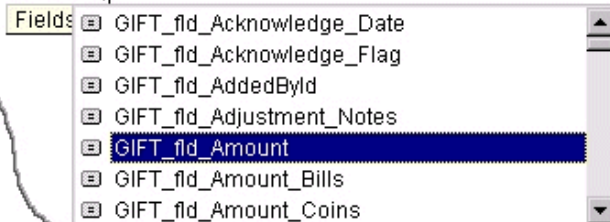
Each data object shares a common and very important property—Fields. The Fields property exposes all the individual, updatable data elements that make up a data object. Instead of exposing a unique property for each field on a data object, which would be cumbersome and very hard to extend, Blackbaud's 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

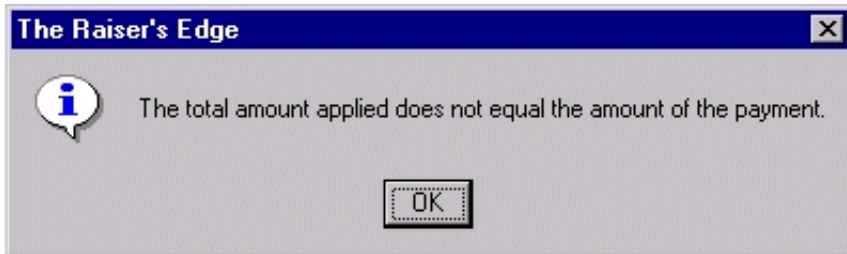
'Save our changes
oGift.Save

'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.



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 Organisation**. 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

'Initialise 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 initialise a new object, set the contents of the **Surname** 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 initialise the parent record. This is the best way to conceptualise 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.

The screenshot shows the 'Mark D Adamson' constituent form. The top menu bar includes File, Edit, View, Constituent, Letter, Favourites, Tools, and Help. Below the menu is a toolbar with icons for saving, printing, and navigating. The main form area has tabs for Attributes, Media, Actions, Honour/Memorial, Volunteer, Prospect, Membership, Events, Bio 1, Bio 2, Addresses, Addressees/Salutations, Relationships, Appeals, Notes, and Gifts. The form contains several dropdown menus for Target (£1,000 - £4,999), Income (£40,000 - £49,999), Receipt type (One receipt per gift), Birthplace (Tennessee), Religion (Catholic), and Ethnicity (Caucasian). Below these is a table for credit cards with columns for Card Type, Credit Card No., and Expiry. A pop-up window titled 'Constituent Codes' is overlaid on the bottom half of the form, showing a table with columns for Description, Code, Date From, and Date To. The table lists Trustee (TR), Volunteer (VOL), Friend (FR), and Major Donor (MD). Buttons for 'Up' and 'Down' are visible on the right side of the pop-up window.

Card Type	Credit Card No.	Expiry
Visa	587740211	04/2006

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

Child objects cannot be created, loaded, saved, initialised, 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 initialise and load a CRecord object (oRecord) omitted for brevity.

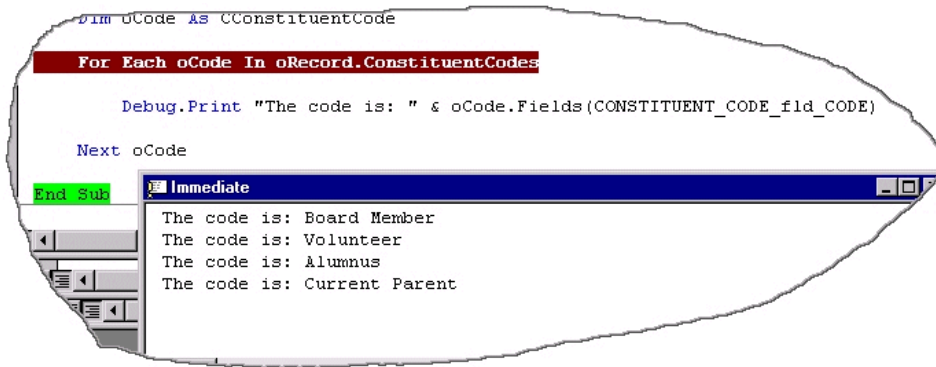
Dim oCode as CConstituentCode

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.



## 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 initialise 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.



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 Favourites Tools Help

Save and Close

General Campaigns Appeals NL Distributions Attributes

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

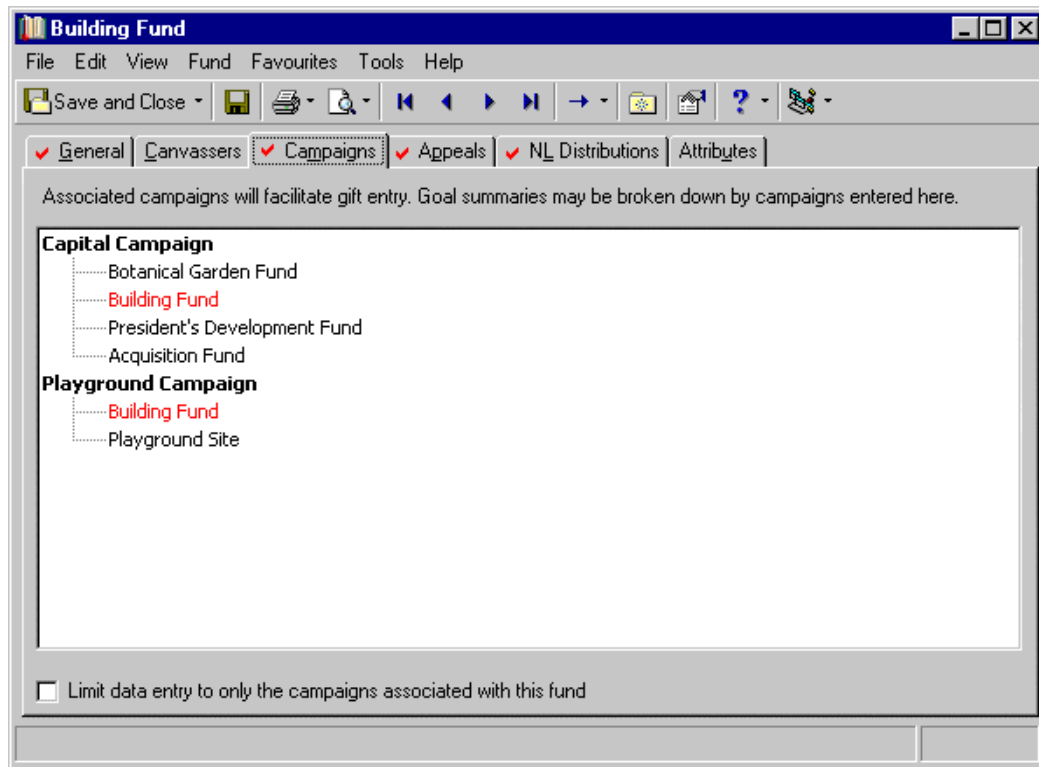
Campaign Description	Campaign ID	Default?
Playground Campaign	PLAYGROUND	<input checked="" type="checkbox"/>
Capital Campaign	CAPITAL	<input type="checkbox"/>
		<input type="checkbox"/>

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

Copy From...



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



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 InstalmentPayments collection exposed by the gift object. Due to the many issues involved in paying off pledge instalments, 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 initialised 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 initialised, 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.

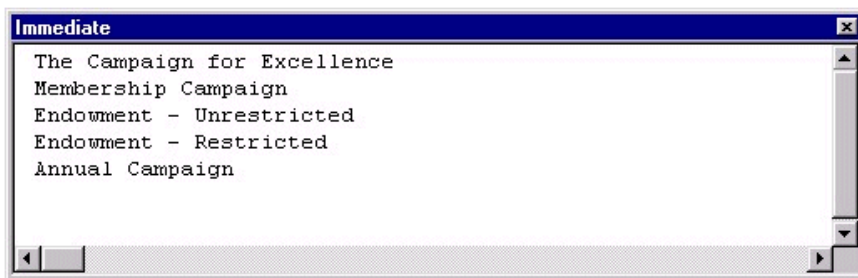
## SortOrder

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

```
'Initialise 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
```



## 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.



## 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 initialised and able to be 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*.

The screenshot shows a Windows-style application window titled "New Campaign". The menu bar includes "File", "Edit", "View", "Campaign", "Favourites", "Tools", and "Help". The toolbar contains icons for "Save and Close", "Save", "Print", "Undo", "Redo", "Find", "Help", and a "New" icon. Below the toolbar are tabs for "General", "Giving Hierarchy", "Canvassers", "Funds", "Appeals", "Attributes", and "Events". The "General" tab is active. It contains the following fields: "Campaign ID:" with the value "LIBRARY"; "Description:" with the value "Library Campaign"; "Start date:" with the value "25/02/2003"; "End date:" which is empty; "Campaign category:" which is a dropdown menu; "Overall goal:" which is a text input field; and a checkbox labeled "Campaign is inactive" which is unchecked. At the bottom is a large "Notes:" text area.

## Raiser's Edge ActiveX Controls

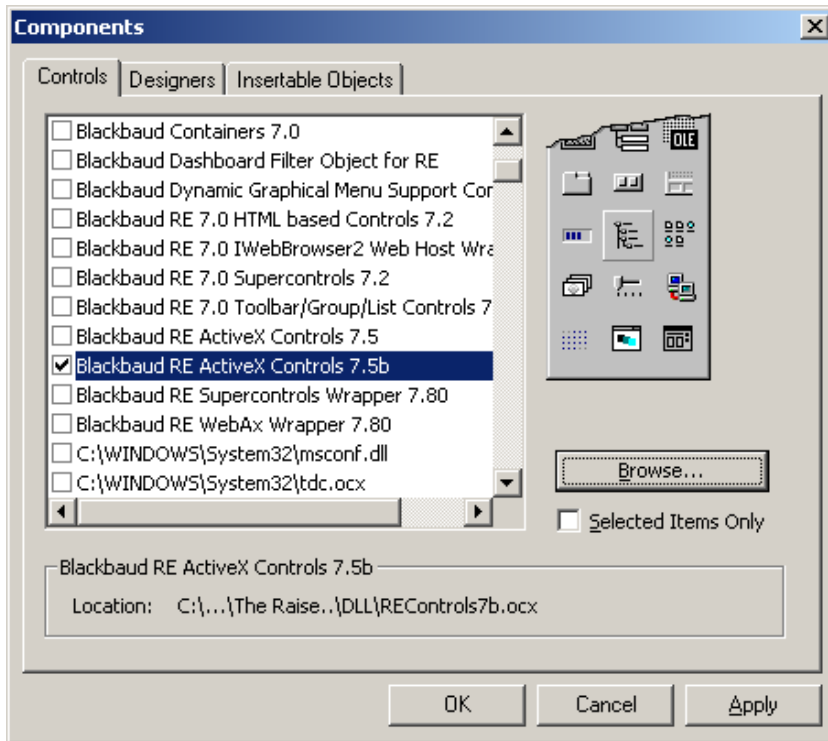
In the previous section, you learned how to use existing *Raiser's Edge* forms in your programs to simplify programming and increase usability. Now, you can also learn how to use a few specific controls to simplify your code. When developing *The Raiser's Edge*, the development team decided to create 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**.



#### 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.

Fund ID	Fund Description	Fund Start date	Fund End date
2000ANNUAL	2000 Annual Fund	01/01/2000	31/12/2000
2000MEMBER	2000 Membership Fund	01/01/2000	31/12/2000
2001ANNUAL	2001 Annual Fund	01/01/2001	31/12/2001
2001MEMBER	2001 Membership Fund	01/01/2001	31/12/2001
2002ANNUAL	2002 Annual Fund	01/01/2002	31/12/2002
2002MEMBER	2002 Membership Fund	01/01/2002	31/12/2002
2003ANNUAL	2003 Annual Fund	01/01/2003	31/12/2003
2003MEMBER	2003 Membership Fund	01/01/2003	31/12/2003
2004ANNUAL	2004 Annual Fund	01/01/2004	31/12/2004
2004MEMBER	2004 Membership Fund	01/01/2004	31/12/2004
2005ANNUAL	2005 Annual Fund	01/01/2005	31/12/2005
2005MEMBER	2005 Membership Fund	01/01/2005	31/12/2005
ACQUISITION	Acquisition Fund	01/01/2000	

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

23 records found.

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 colours, 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

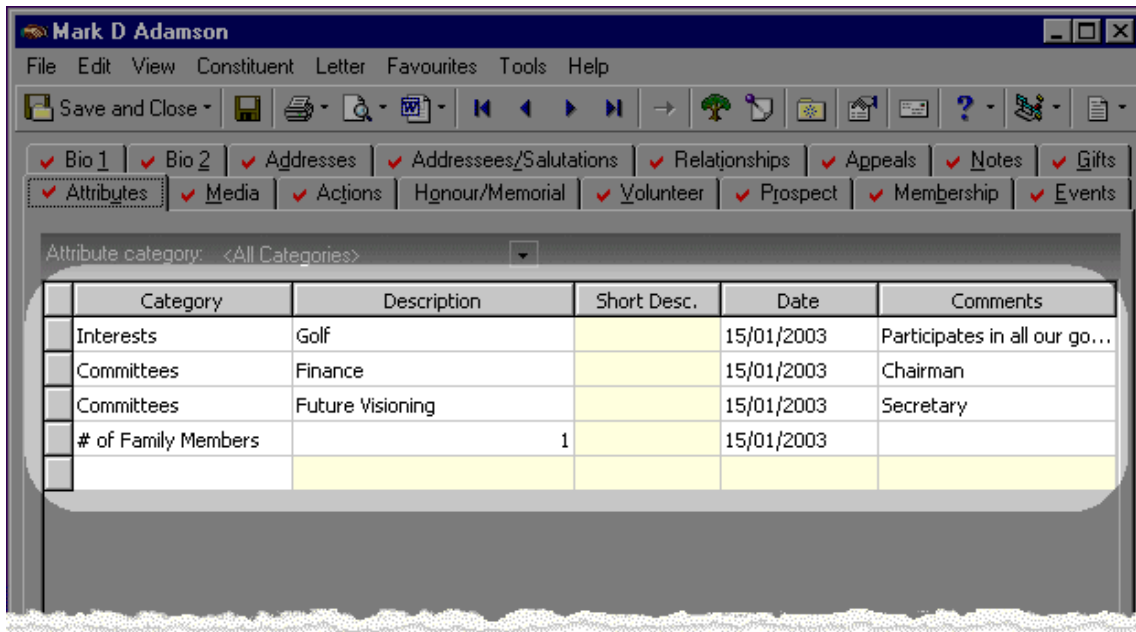
    'Clean up!
    oGift.CloseDown
    Set oGift = Nothing

    oGiftForm.CloseDown
    Set oGiftForm = 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.





To use the grid, you must first initialise 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 IBBAAttribute 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

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*.

The screenshot shows a data entry form for 'Mark G Adamson'. The form is divided into several sections. On the left, the 'Biographical' section contains fields for Surname (Adamson), First name (Mark), Middle name (G), Titles (Mr), Suffixes, Nickname, Maiden name, ID (1234), Gender (Male), Birth date (23/09/1954), Age (50), Deceased? (checkbox), Date, Marital status (Divorced), Spouse (Mary Adamson), and Solicit code (Do not phone). On the right, the 'Preferred Address : Home' section contains fields for Country (United Kingdom), Address lines (125 Summerville Way), City (Bristol), County, Postcode (BS2 8DJ), and DPS. Below this is the 'Phones/Email/Links' grid, which is a table with columns: DNC, Type, Number/Email Addr..., Shared?, and an empty column. The grid contains one row: Home, 01234 124 5689, Not Shared. At the bottom right, there is a section for 'This constituent' with checkboxes for 'Is inactive', 'Is a canvasser' (checked), 'Gives anonymously', 'Has no valid addresses', 'Requests no e-mail', and 'Declaration Information' (checked). The bottom of the form has buttons for 'Education', 'Business' (checked), and 'Bank' (checked).

DNC	Type	Number/Email Addr...	Shared?	
	Home	01234 124 5689	Not Shared	

## 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 categorised because they each provide a service via their own unique programming interface. To help organise 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 initialise 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 <b>Description</b> for the new query.
lSystemID	Long	Included in parameters, but not applicable.
sDefaultQName	String	Optional: Allows you to input a default <b>Query name</b> 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 surnames 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

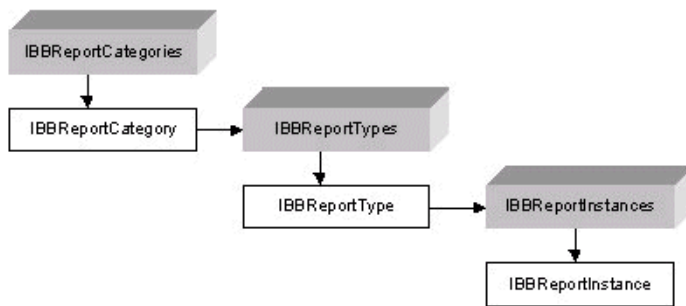
oRecord2.Init oAPI.SessionContext
oStaticQuery.Init oAPI.SessionContext
'This will prompt the user for a Query Name but
' everything will already be filled in
If oStaticQuery.Create(SEARCH_CONSTITUENT, "Custom App", Nothing, _
    "List of couples who have different last names", , _
    "Spouses W\Different 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 information, see the following "Report Objects Sample" on page 55). After you create an IBBReportCategories collection, use the Init method. This is similar to the Init method used for other objects to initialise 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 initialise 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 <i>Reports</i> , <i>Mail</i> , 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 <i>Reports</i> , such as query control reports or <b>Global Add</b> reports. If this is set to True (the default), only the IBBReportCategory objects that represent categories found in <i>Reports</i> are in the collection.

When you initialise 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 initialise all objects.
CategoryID	EReR_ReportCategories	This is a Enum of all the different categories of reports using in the <i>Raiser's Edge</i> .

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 <i>Reports</i> . If set to True (which is the default), only the IBBReportType objects that represent reports found in <i>Reports</i> is included in the IBBReportCategory.ReportTypes collection.

When you initialise 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 55 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 55). 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 initialise 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 <i>Reports</i> . If this is set to True (which is the default), only the IBBReportType objects that represent reports found in <i>Reports</i> are included in the IBBReportCategory.ReportTypes collection.

After initialising 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 initialise 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 initialised, 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 55). When you create the object, use the `Init` method to initialise it. The following table shows the parameters for the `Init` method.

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

After you initialise 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
<code>SessionContext</code>	<code>IBBSessionContext</code>	This is the same <code>SessionContext</code> used to initialise all objects.
<code>QueMode</code>	Boolean	Optional: This establishes how errors are addressed when using the object. If set to <code>True</code> , a log file is created containing any errors, but the program continues to process. If set to <code>False</code> (which is the default) a trappable error is raised.

After you initialise 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
<code>Action</code>	<code>EReR_ProcessOptions</code>	This is an Enum of the process options available.
<code>ShowModal</code>	Boolean	Optional: This determines if the <code>Process</code> form (this varies depending on the action) is displayed modally. This defaults to <code>False</code> .
<code>FormToCenterOn</code>	Object	Optional: This determines which object the <code>Process</code> form displays.

Parameter	Variable Type	Description
RunFromWeb	EReR_WebReportType	Optional: This is used internally for the <i>The Raiser's Edge for the Web</i> . 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, minimises 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 initialisation 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 59 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 <i>The Raiser’s Edge</i> .
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 <i>The Raiser’s Edge</i> , 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.

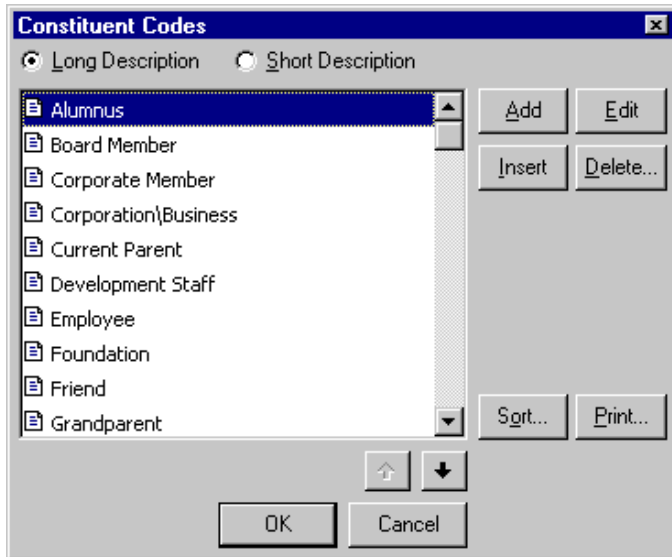
```
Dim lLong As Long
Dim sString As String

'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 57 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 initialisation 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
ICodeTableID	ECodeTableNumbers	This is the ID for the particular code table you want to display.
IFindItemData	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 61 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 63 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 Combol_LostFocus()

    Dim sMsg as String

    If Len(Combol.Text) > 0 Then

        With oCodeTablesServer

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

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

                If MsgBox(sMsg, vbQuestion + vbYesNo) = vbYes Then
                    'This adds the current text to the database and
                    ' Refreshes Combol. If the AddEntry is unsuccessful
                    ' this will return False.
                    If Not oTableLookupHandler.AddEntry(True, _
                        tbnunMaritalStatus, , Combol.Text, combol) 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.
                    Combol.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 42 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 57 and "Table Lookup Handler" on page 60.

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 initialised, 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 57 and “Table Lookup Handler” on page 60 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
                Comb1.Visible = True
                Comb1.AddItem "Yes"
                Comb1.AddItem "No"

            Case bbAttribute_TABLEENTRY
                Comb1.Visible = True
                'This uses the CodeTablesServer to the load the combo
                '    with all of the table entries
                moCodeTablesServer.LoadCombo Comb1, _
                                           .GetAttributeCodeTableID(lAttribute_ID), , True

            Case Else
                Text1.Visible = True

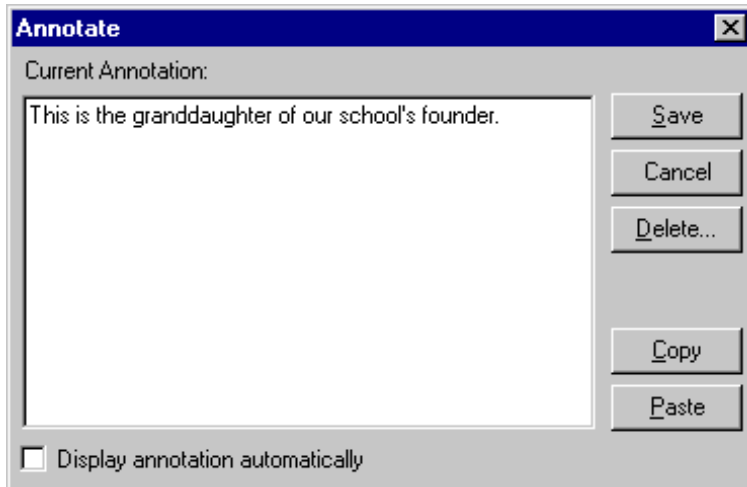
        End Select

    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.



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 initialise 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. Initialise 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.

## 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 initialise 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. Initialise 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 <b>File</b> menu are: <b>Save</b> , <b>Save &amp; Close</b> , <b>Properties</b> , and <b>Close</b> .
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.

The screenshot shows a Windows-style dialog box titled "New Media for Joseph A. Mahoney". It features a menu bar with "File", "Edit", "View", "Media", and "Help". Below the menu is a toolbar with icons for "Save and Close", a disk icon, a close icon, navigation arrows, a print icon, and a help icon. The form contains several input fields: "Date" with the value "31/03/2004", "Title" with "Golf Winner", "Type" with a dropdown menu showing "Picture", "Author" with a dropdown menu showing "Supervisor", "Proposal" with an empty text box, and "Description" with the text "Winner of the 2004 golf tournament". At the bottom of the dialog is a large image area displaying a photograph of a golfer in mid-swing.

### 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 initialise 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. Initialise 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

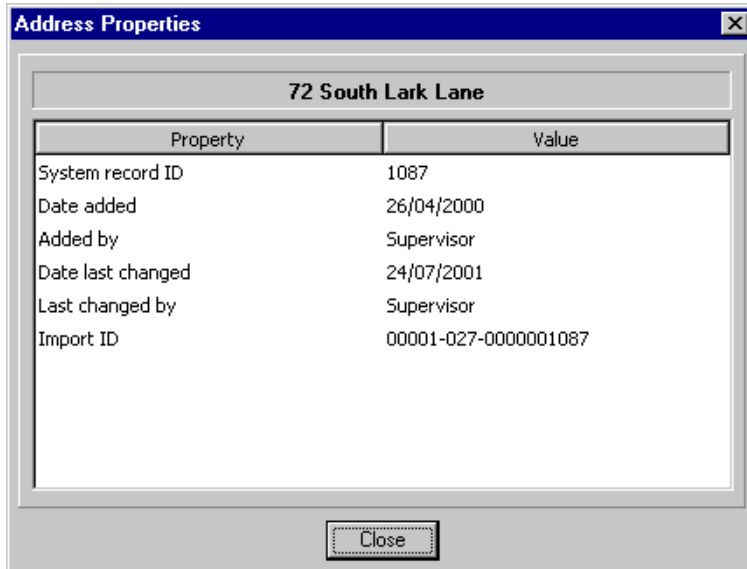
'The user's changes are not added to the database until this is called.
oRecord.Save

'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.



The screenshot shows a window titled "Address Properties" with a close button (X) in the top right corner. Inside the window, there is a header section labeled "72 South Lark Lane". Below this header is a table with two columns: "Property" and "Value". The table contains the following data:

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

At the bottom of the window, there is a "Close" button.

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 initialise 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. Initialise 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 initialise 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. Initialise 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 programming. 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
    End With

    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 `PromptForObject` 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 initialise 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. Initialise 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 `PromptForObject`. 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
<code>lSearchType</code>	<code>bbSearchTypes</code>	This is an enum of the record types available for searching.
<code>sObjectNameCaption</code>	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.
<code>sDescriptionCaption</code>	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. <i>Note:</i> <b>Quick Find</b> 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 IBBDDataObjectInterface, 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 IBBDDataObject 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 IBBDDataObject interface. This provides a way for you to refer to any data object that implements IBBDDataObject in a generic fashion. Referring to the IBBDDataObject interface gives you access to some properties and methods that are not available when referring to the actual class.

By referring directly to the IBBDDataObject interface, you can use the Initialised property to find out if an object has been initialised. 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 IBBDDataObject 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 IBBDTopObject 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 IBBDTopObject)

    Dim oIBBDDataObject As IBBDDataObject
    Set oIBBDDataObject = oTopObject

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

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

    If oIBBDDataObject.Dirty Then oTopObject.Save

    'Clean Up!
    Set oIBBDDataObject = 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 IBBDDataObject Interface” on page 81 interface to return a reference to the IBBMetaField interface.



## Code Sample

```
Option Explicit

Private moAction As CAction
Private moMetaField As IBMetaField
Private moDataObject as IBDataObject

Private Sub Form_Load()

    Dim i As Integer

    Set moAction = New CAction
    moAction.Init REApplication.SessionContext

    'Provides access to the IBMetaField 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.
            Labell(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
                Labell(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 Labell_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 customised view of individual and organisation constituent records. This makes it possible to view records showing only the information you want to see. By creating these customised screens, you can select and view only what is important and functional for your organisation. Every field available on a constituent record may not be useful to you. Using *Custom View*, you can easily select and organise 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 customise parts of custom screens to fit the particular needs of your organisation. 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 colour 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 colour 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 oHTMLElement 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.
    oHTMLElement.innerHTML = "<SPAN style='font-size:20pt;_
colour: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;colour: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

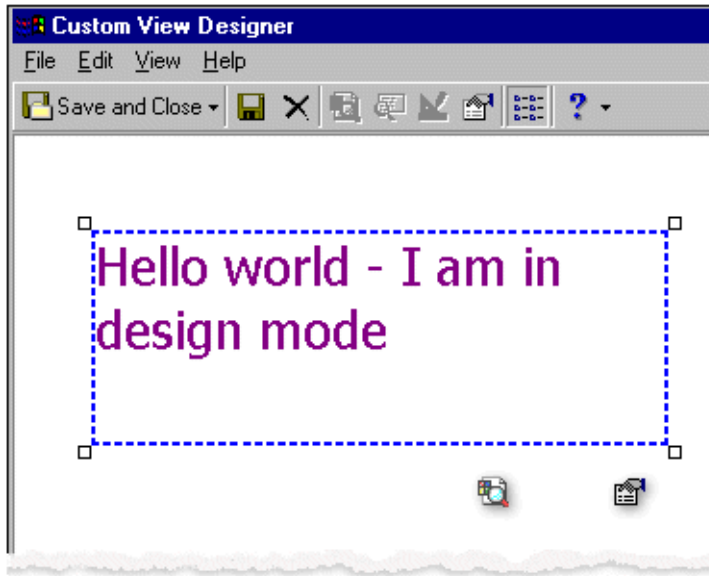
        oGift.CloseDown

    Next oGift

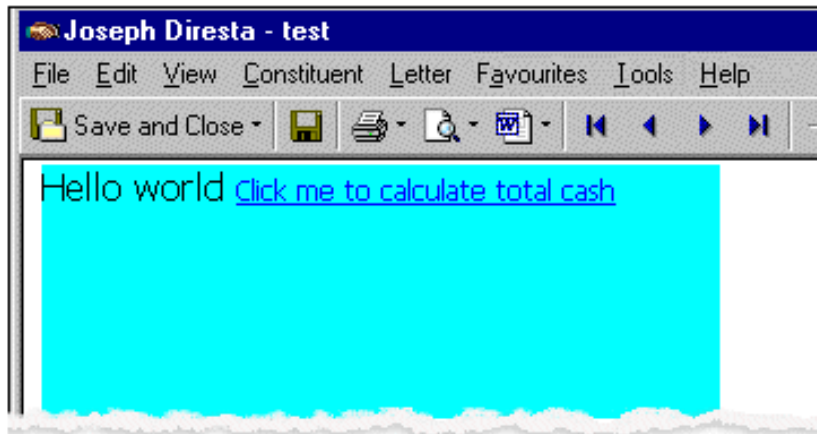
    moDIV.innerHTML = "<SPAN style='font-size:20pt;font-family:impact;colour:green'>_
Total Cash:" & FormatCurrency(cTotal) & "</SPAN>"

End Sub
```

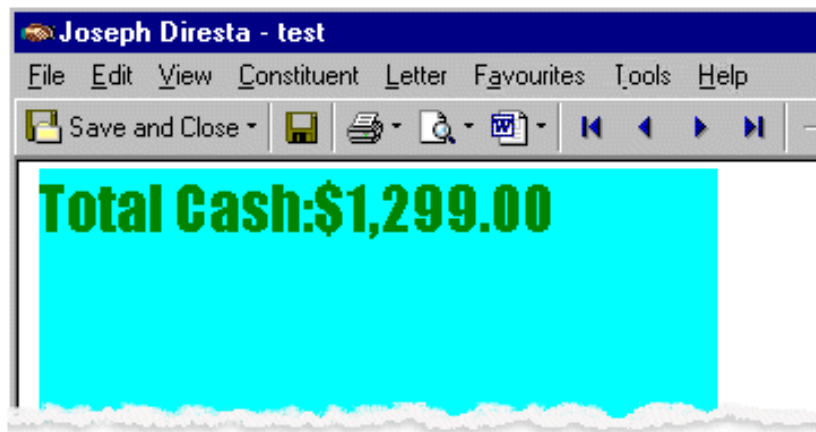
To use the new custom part, click **Configuration** in the Raiser's Edge bar and click the **Custom View** link. In Custom View, custom parts are available from the Fields screen, at the bottom of the **Additional Objects** box. When you drag our sample into a custom view you are creating, here is what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



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.



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



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



# VBA

## Contents

<b>What is VBA?</b> .....	<b>96</b>
<b>API vs. VBA</b> .....	<b>97</b>
<b>The VBA Environment.</b> .....	<b>97</b>
Accessing the VBA Environment. ....	97
Opening the IDE .....	98
A Quick Tour .....	98
The Project Window .....	99
The Code Window. ....	99
The Properties Window. ....	99
The Forms Designer .....	100
A Note About Saving Your Code .....	100
<b>Macros</b> .....	<b>101</b>
Types of Macros .....	101
Active Object Macros and The Raiser's Edge Code Wizard .....	102
Creating the Active Object Macro. ....	102
Debugging the Active Object Macro. ....	103
Running the Active Object Macro. ....	103
Standard Macros .....	103
Data Object Macros. ....	105
Query and Export Macros .....	107
Macro Samples .....	108
Setting Defaults .....	109
Adding Notepad Records .....	110
The Active Objects .....	111
The REApplication Object .....	112
The Active Data Objects .....	113
Active Process Objects .....	121
<b>VBA DLL Macros.</b> .....	<b>140</b>
The DLL Tool .....	140
Creating DLL Macros .....	140
Debugging DLL Macros .....	141
Running DLL Macros .....	142
<b>Customising RE:Anywhere for Remote Access Using RE:VBA</b> .....	<b>142</b>
The RE:Anywhere for Remote Access VBA Object Hierarchy .....	143
VBA DLL Macro Permissions and Memory .....	144
Debugging the VBA DLL Macro and Run-Time Errors .....	144
Sharing Code Between RE:Anywhere for Remote Access and The Raiser's Edge	146
<b>VBA Code Samples.</b> .....	<b>147</b>

# Chapter 2

This section introduces the VBA implementation in *The Raiser's Edge*. With the optional module *VBA for Advanced Customization*, the experienced developer can use macros to leverage the power of *The Raiser's Edge* programmatically from within the program shell. This section builds upon many concepts introduced in other sections of this guide. Be sure to familiarise yourself with the material discussed in the Essentials chapter.

#### 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.co.uk](mailto:solutions@blackbaud.co.uk).

## What is VBA?

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

### VBA vs. API

This section explains how API differs from VBA support. It is important you understand which customisation methodology is appropriate for your task.

## The VBA Environment

This section introduces the *Visual Basic* development environment.

### Macros

This section discusses the different types of macros, how they are built, and how they are executed.

### VBA DLL Macros

In addition to macros contained in *The Raiser's Edge*, you can create Dynamic Linking Library's (DLL) containing macros and distribute the compiled macros rather than the source code.

## VBA Code Samples

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

#### For more information...

Visit Blackbaud's Web site at [www.blackbaud.co.uk](http://www.blackbaud.co.uk) for software customisation 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 customising your *Raiser's Edge* software. You can also contact Customer Support for assistance.

## What is VBA?

Although this guide assumes you are somewhat familiar with VBA, a short explanation of what *Visual Basic for Applications* is and how it can be used within *The Raiser's Edge* helps you get started. For more information about the applications, see the Essentials chapter.

*Visual Basic for Applications* is the premier development technology for rapidly customising and integrating packaged applications. *VBA* offers a sophisticated set of programming tools based on Microsoft *Visual Basic*, enabling developers to create solutions meeting specific business needs in much less time than it would take to build them from scratch.

You can use *VBA* to enhance and customise *The Raiser's Edge* to meet your organisation's needs in countless ways:

- Modifying application behavior. Modifying the way *The Raiser's Edge* works to match your organisation's business rules and processes.
- Automating repetitive tasks. Combining sets of common manual tasks into a series of actions that can be executed over and over.
- Extending application functionality. Adding features to *The Raiser's Edge* that are not available out of the box.
- Integrating with other applications. Working seamlessly with other *VBA*-enabled software to integrate a line of business applications.
- Accessing data. Exchanging data with remote databases and applications and delivering results directly to the desktop.

## 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 program, or create a customised system that melds *The Raiser's Edge* with some other specialised 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 program in Microsoft *Excel* that adds records to *The Raiser's Edge*. Using this example, a Microsoft *Excel VBA* macro uses *API for Advanced Application Development* to add records to *The Raiser's Edge*.

## The VBA Environment

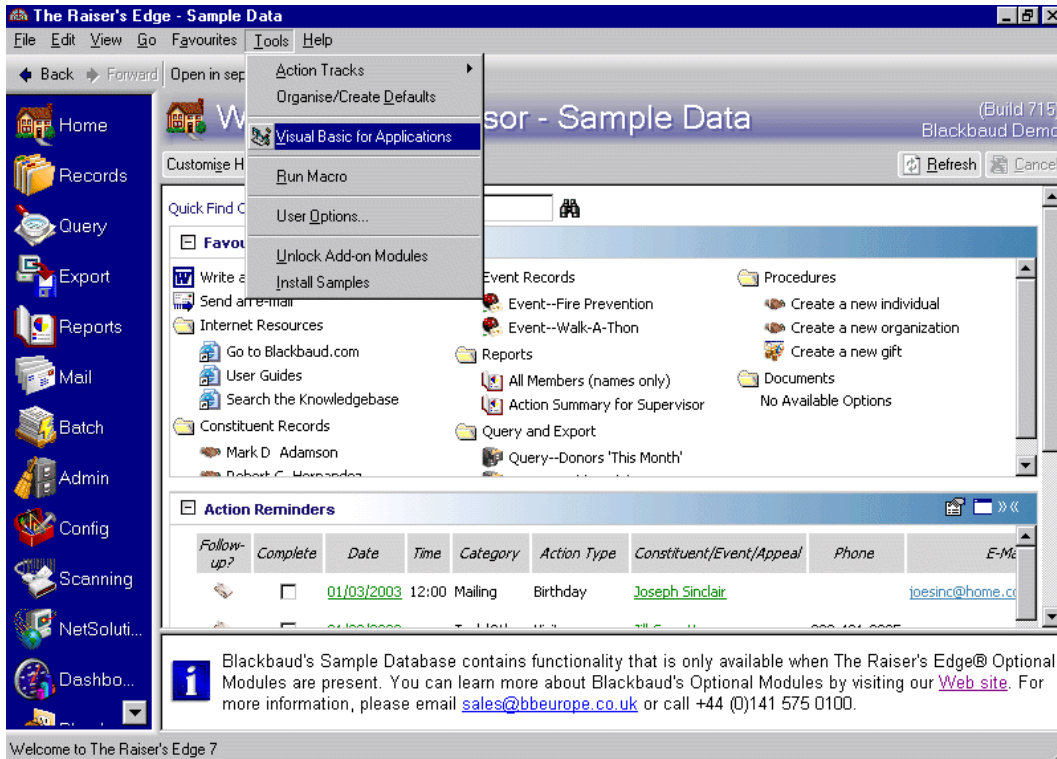
This section introduces the steps involved in *VBA*. The *VBA* environment (also referred to as the IDE) available within *The Raiser's Edge* represents the standard environment provided by any *VBA*-enabled application. Part of *VBA*'s appeal is that it has the same look and feel regardless of the *VBA* host application you start it from.

## Accessing the VBA Environment

The *Visual Basic* programming environment is easily accessible from within *The Raiser's Edge*. An important concept to understand is that *VBA* is hosted by *The Raiser's Edge*. *VBA* is not a standalone application. It is an integrated component of *The Raiser's Edge*. Many other applications host their own instances of *VBA*. For example, Microsoft *Excel* and *Word* each embed *VBA*. While all these hosts (*Raiser's Edge*, *Word*, *Excel*) share the common *VBA*-integrated development environment (IDE), each exposes its own custom functionality and documents as project items.

## Opening the IDE

To access the VBA IDE, launch *The Raiser's Edge*. From the **Tools** menu, select **Visual Basic for Applications**. Microsoft *Visual Basic* launches and displays on your screen.

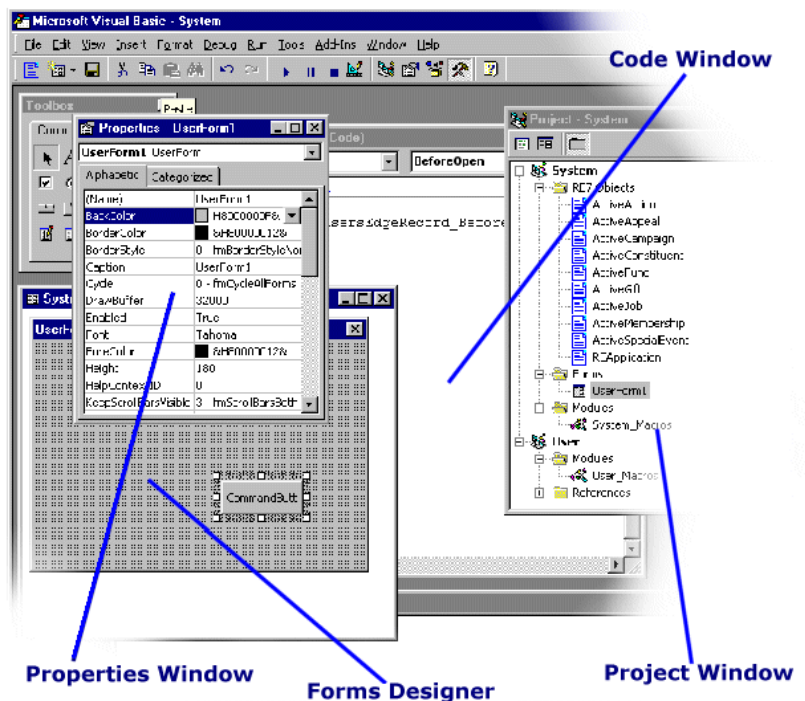


## A Quick Tour

The integrated programming environment runs in its own window. Advanced debugging features, property and code editing features (including compile time syntax checking) an enhanced object browser, and code organisation and tracking make the VBA IDE a powerful platform from which to develop code.



This section provides an overview of some of the major features of VBA and how they apply to programming *The Raiser's Edge*.



## The Project Window

Also known as the “Project Explorer,” the project window organises all VBA code stored within *The Raiser's Edge*. *The Raiser's Edge* has two default Project Items, System and User.

The System project holds references to active instances of *Raiser's Edge* objects and contains code that executes for all users. This project is the place to build your custom business rules and functions. Only the supervisor can edit code in the system project. The code executes for other users, but they are not able to modify it.

The User project is provided to give individual end-users a project in which to save their individual code and macros. Code stored in the User project is available to a specific user and helps keep his code organised and avoids cluttering or compromising important system code.

## The Code Window

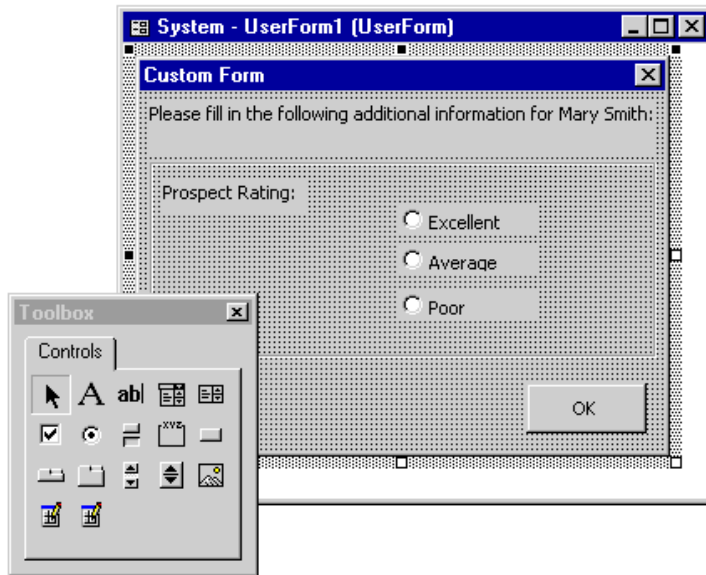
To view the code in a project, open the Project Explorer. Click the element containing the code and then click the **View Code** button at the top of the Project Explorer. The code displays in the code window, where it can be edited and debugged.

## The Properties Window

A property is an attribute of an object, such as the object's colour or caption. You set a property to specify a characteristic or behavior of an object. You use the properties window to define the properties of an object at design time. This window is useful when building custom dialog boxes and forms using VBA's forms designer.

## The Forms Designer

VBA provides full-featured forms design support. You can call these forms from VBA code you write to execute within *The Raiser's Edge*. For example, you can design a custom data entry form that appears when an end-user saves a constituent record.



### A Note About Saving Your Code

*The Raiser's Edge* stores code written in the optional module VBA in the *Raiser's Edge* database. If you restore from a backup, the code in the restored database is the code written up to the time when the backup was performed. Any code written after that time is not included in the restored database.

If you install a new database (for example, you may be using a test database during the conversion process), it does not contain any of your code at all.

If you do not perform nightly backups, or you need to install a new database, you may want to consider saving your code outside *The Raiser's Edge* database. You can do this by either exporting the code (which you can import back into the database later), or copying and pasting the code into a text file.

#### ➤ To export code:

1. Select the instance you want to export (for example, Active Constituent).
2. From the menu bar, select **File, Export File**.
3. Name the file and select the location you want to save it to.

#### ➤ To import code:

1. From the menu bar, select **File, Import File**.
2. Browse to the file you want to import.
3. Select the file and click **OK**.

# Macros

If you perform a task repeatedly in *The Raiser's Edge*, you can automate the task with a macro. A macro is a series of steps stored in a VBA module or VBA DLL, that can be run whenever you need to perform the task.

*The Raiser's Edge* provides two methods for creating and editing macros. The first method is accessed through *The Raiser's Edge* shell from the **Tools, Visual Basic for Applications** menu bar.

The second method uses a program separate from *The Raiser's Edge*, RE7VBA.EXE. The process of creating macros through either means is similar, but each method has its own advantages. For our purposes, macros created through *The Raiser's Edge* shell are referred to as VBA Macros. Those created through the RE7VBA.EXE program are referred to as VBA DLL Macros.

## Types of Macros

Through VBA you can create four different types of macros.

### Active Object Macros

With active object macros, you can take control of a top level object or process at key points in its lifetime. Standard events occur for each data object as it is loaded, saved, closed, and deleted. Other events fire at the beginning and end of certain processes. You can write code inside these events to take action on a given data object, to enforce custom business rules, or to prevent processes from occurring under specified conditions.

For more information about active object macros, see “The Active Objects” on page 111.

### Standard Macros

With standard macros, you can perform specific functions from the program shell. For example, you could write a macro that automates the printing of a set of End of Day, End of Month, or End of Year reports. You first create the reports for each group, and then create the macro to automate the printing process. Your end-users can select the macro and run all the reports with the click of a button.

For more information about standard macros, see “Standard Macros” on page 103.

### Data Object Macros

With data object macros, you can run a set of instructions using the data object with which you are currently working. All top level objects and some commonly used forms have the option to run a data object macro. When you run the macro, VBA is called with a data object representing the current record so you can manipulate that record and navigate its object model. You could use data object macros to give your end-users a quick summary of a constituent's giving history, displaying the information in a format of your own design.

For more information about data object macros, see “The Active Data Objects” on page 113.

### Query and Export Macros

Query and Export macros provide you with access to each result row as the query or export is processed. With the use of special VBA user fields, you can modify the query or export results. For example, you could include a VBA user field that is the sum of two other fields in your query.

For more information about query and export macros, see “Query and Export Macros” on page 107.

## Active Object Macros and The Raiser's Edge Code Wizard

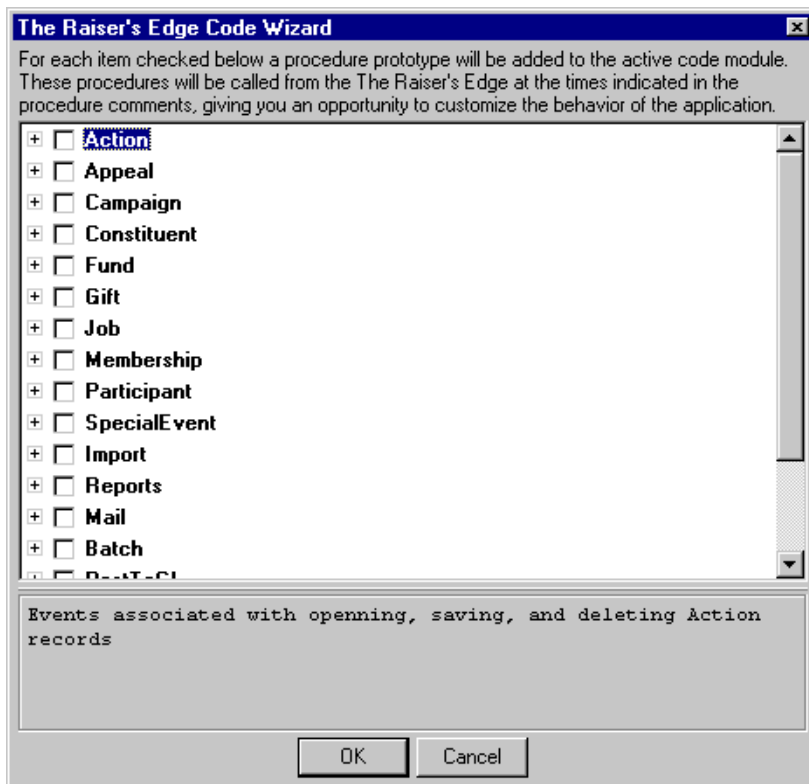
Active object macros are executed during certain events in an object or process life cycle. For example, you can enter code that is executed before a constituent record is saved. If the constituent does not meet custom business requirements, you can cancel the save event.

### Creating the Active Object Macro

To enter the VBA IDE, select **Tools, Visual Basic for Applications** from the menu bar. Then, click **RE Code Wizard** from the toolbar. The Raiser's Edge Code Wizard screen appears.

#### Please remember....

Only the Supervisor can create Active Object Macros.



The code wizard simplifies the process of creating Active Object macros. After selecting a set of events to respond to, the code wizard generates code that includes useful comments and error checking. For example, if you need to enforce a rule that all constituent IDs are 10 digits long you can follow these steps to enforce the business rule.

1. Select "Constituent" from the record type list.
2. Mark **Constituent\_BeforeSave**. Uncheck the other project checkboxes.
3. The code appears in the VBA window.

4. The comments explain the parameters that are available. “oRecord” is the project record being saved. The record has not been stored in the database, but the oRecord object contains all information defined for the unsaved record. oGLProject is the class that manages projects in *The Raiser’s Edge*. When you set oGLProject to oRecord you can use Intellisense. With the bCancel parameter you can cancel the save event. Add the following code after the “If statement”.

```
If  
  Len(oConstituent.Fields(RECORDS_fld_CONSTITUENT_ID)) <> 10 Then  
    MsgBox "Constituent IDs must be 10 digits"  
    bCancel = True  
  End If
```

5. Click **Save** and test the macro. Enter a constituent with a 12 digit constituent ID. A message appears and the save event is canceled.

Active objects are divided into three groups:

- Data objects include all the top level objects in the list. Some examples are ActiveAction, ActiveAppeal, ActiveCampaign, ActiveConstituent, ActiveFund, ActiveGift, ActiveJob, ActiveMembership and ActiveSpecialEvent. These objects have events tied to opening, closing, saving, and deleting a record.
- Process objects include ActiveBatch, ActiveImport, ActiveMail and ActiveReport. These objects have events for starting and ending a process. In addition, Import and Batch allow you to interact with each record as it is validated.
- The REApplication object which represents the application. Here, you can write code that occurs when the application is opened or closed. Use the bCancel parameter to cancel the close event of *The Raiser’s Edge*. The available events vary based on the type of object selected. For more information about specific objects, see “The Active Objects” on page 111.

## Debugging the Active Object Macro

Because VBA is integrated into *The Raiser’s Edge* shell, you can debug your macros while the application is running. You can place breakpoints into the macro code and use all the standard debugging tools provided with the VBA IDE.

## Running the Active Object Macro

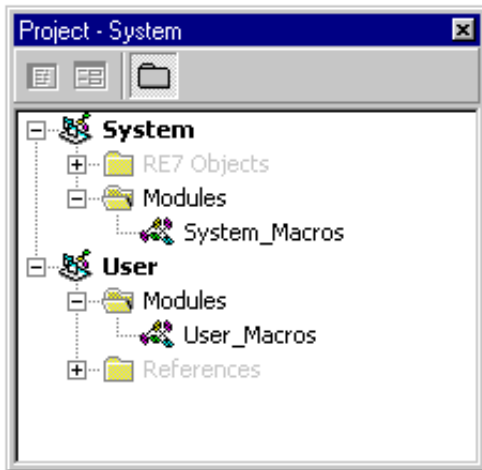
Active Object macros occur automatically when the end-user takes an action to cause a specific event. Once written, the macro fires for each end-user whenever the event occurs.

## Standard Macros

A standard macro is a block of code you can execute from anywhere in *The Raiser’s Edge* shell. Creating macros for your end-users opens the door to a wide range of options that simplifies work and increases efficiency.

## Creating a Standard Macro

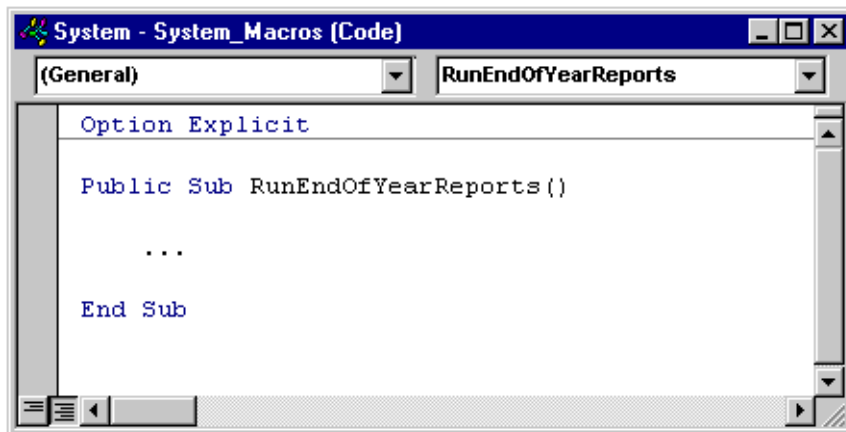
To enter the VBA IDE, select **Tools, Visual Basic for Applications** from the menu bar. The Project Explorer has two projects, System and User. Under each of these projects is a module, System\_Macros and User\_Macros, respectively. You can place your macros into these modules, or add additional modules as needed.



You can create macros in the System\_Macros module in the System project only when you are logged into *The Raiser's Edge* as Supervisor. These macros are available to all users. However, within your macro code you can access the current user name and limit who can actually run the macro.

Macros written in the User\_Macros module in the User project are available only to the user who created them.

Next, create a public subroutine in the System\_Macros or User\_Macros module. Standard macros cannot have any parameters. Keep in mind that the name of your macro is displayed to your end-users, so it should be clear from the name what the macro does.

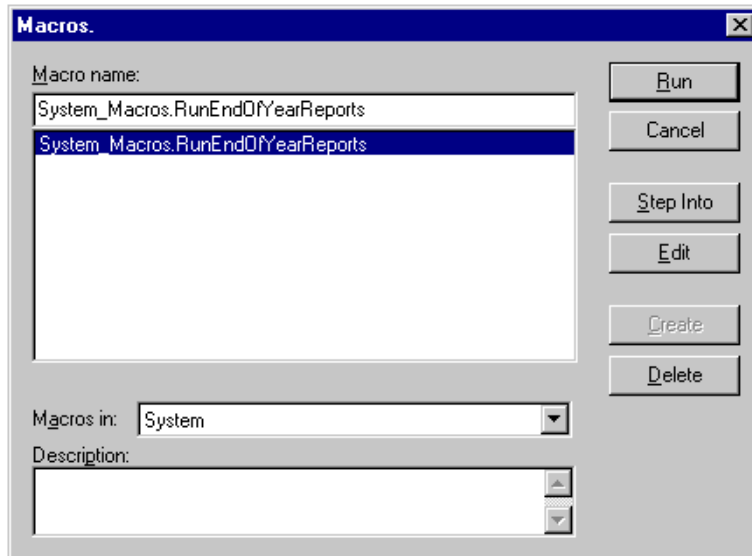


## Debugging a Standard Macro

Since VBA is integrated into *The Raiser's Edge* shell, you can debug your macros while the application is running. You can place breakpoints into the macro code and use all the standard debugging tools provided with the VBA IDE.

## Running a Standard Macro

Run standard macros from the **Tools, Run Macro** menu bar in *The Raiser's Edge* shell. This is how your end-users access and run the macros.

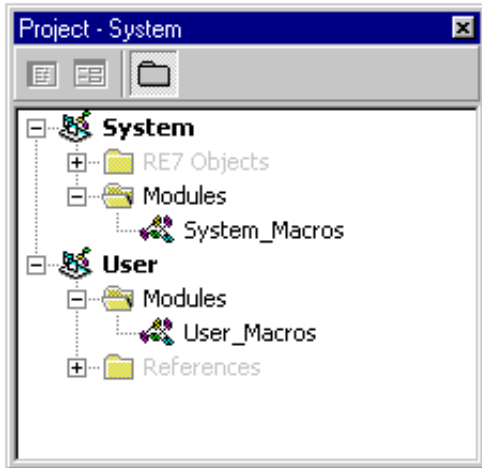


## Data Object Macros

With data object macros, you can run a set of instructions using the data object with which you are currently working. All top level objects and some other commonly used forms, such as Notepad and Address, have the option to run a macro using the underlying data object. When the macro is run, *VBA* is called with the live data object representing the current record, enabling you to navigate its entire object model. Through data object macros, you can write a calculation to fill in a data field, automatically add a list of attributes to a constituent, or fill in default values for a record based on user input. To create or edit a data object macro, you must log in as Supervisor or as a member of the Supervisor group.

## Creating the Data Object Macro

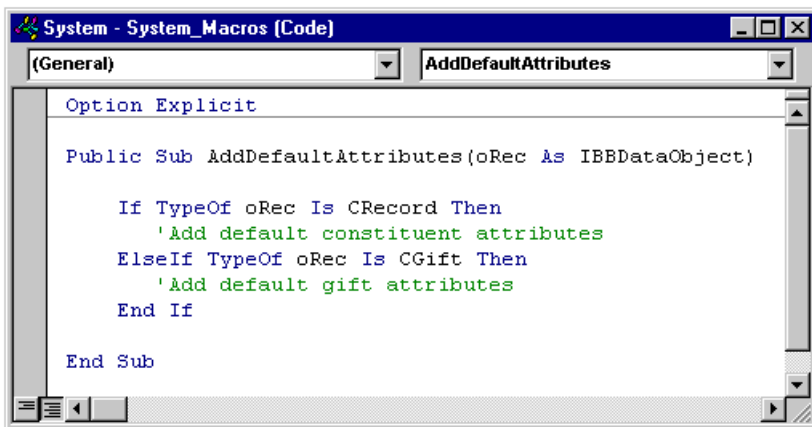
To enter the VBA IDE, select **Tools, Visual Basic for Applications** from the menu bar. Next, open the Project Explorer (Ctrl+R); it has two projects, System and User. Under each of these projects is a module, System\_Macros and User\_Macros, respectively. You must place your macros into these modules for them to be recognised by *The Raiser's Edge* as data object macros. Additional modules can be added as needed to provide support for your System or User macros.



You can create System\_Macros macros only when you are logged into *The Raiser's Edge* as Supervisor. These macros are available to all users. However, within your macro code you can access the current user name and limit who can actually run the macro.

Macros written in the User\_Macros module in the User project are available only to the user who created them.

Next, create a public subroutine in the System\_Macros or User\_Macros module. Data object macros must have 1 parameter of type *IBBDataObject*. Without this parameter, the program does not recognise your macro as a Data object macro.



Data Object macros are available from any data object supporting a data object macro. For example, even though a macro is designed for the constituent data object, it is still be available when the end-user looks at the macro list from a gift record. Therefore, you need to make sure that data object being passed in is of the correct type for that particular macro.



## Debugging the Data Object Macro

Because VBA is integrated into *The Raiser's Edge* shell, you can debug your macros while the application is running. You can place breakpoints into the macro code and use all the standard debugging tools provided with the VBA IDE.

## Running the Data Object Macro

Data object macros are made available to the end-user from the **Select Data Objects, Macros** menu bar.

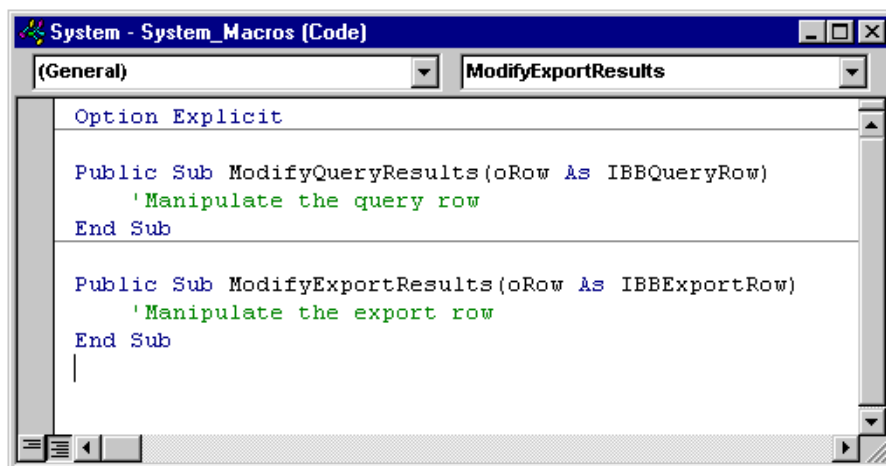
## Query and Export Macros

Using Query and Export macros, you can detect the start and end of the query or export process and modify the results through the VBA user fields. Once written, the macros are attached to the query or export.

## Creating the Query and Export Macro

To enter the VBA IDE, select **Tools, Visual Basic for Applications** from the menu bar. Next, load the Project Explorer (Ctrl+R); it has two projects, System and User. Under each of these projects is a module, System\_Macros and User\_Macros, respectively. Query and Export macros must be written in either of these modules to be recognised by *The Raiser's Edge*.

Next, create a public subroutine in the System\_Macros or User\_Macros module. Query macros take one parameter, IBBQueryRow and Export macros must have one parameter of type IBBExportRow. Without these parameters, your macro is not recognised as a query or export macro.

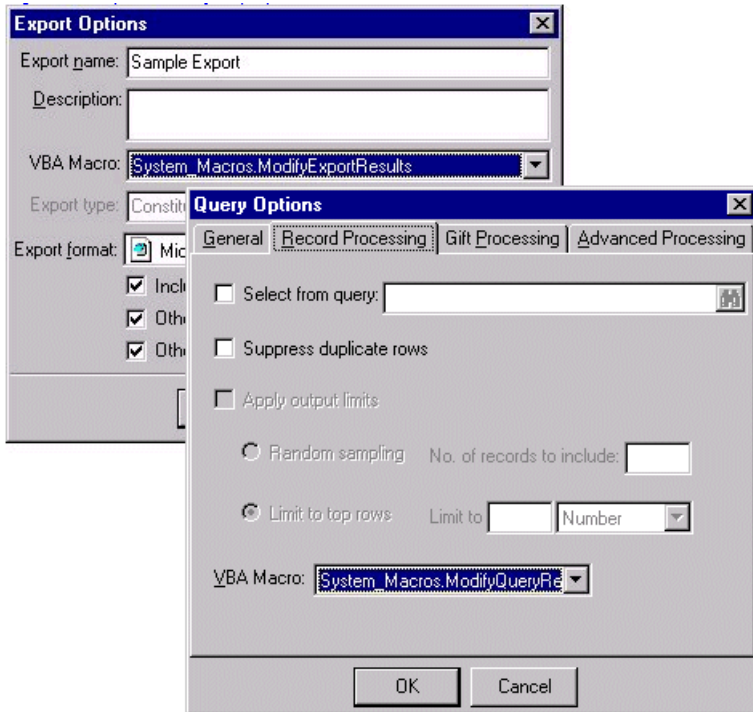


## Debugging the Query and Export Macro

Because VBA is integrated into *The Raiser's Edge* shell, you can debug macros while the application is running. You can place breakpoints into the macro code and use all the standard debugging tools provided with the VBA IDE.

## Running the Query and Export Macro

Export macros are attached to export records through the Export Options dialog. From the Export menu bar, select **Tools, Export Options**. Select the macro you want in the **VBA Macro** field. Query macros are attached in a similar manner. From the Query menu bar, select **Tools, Query Options**. Next, select the **Record Processing** tab, and choose the macro you want in the **VBA Macro** field.



When a macro is attached to a query or export, the macro is fired for each row in the result set, plus once when the process begins and again when the process ends.

## Macro Samples

The following two code samples demonstrate basic macro concepts. The first, “Setting Defaults” on page 109 is a data object macro that sets the default phone type for a record depending on its key indicator. The second, “Adding Notepad Records” on page 110, uses a number of VBA objects and features. It is a standard macro that prompts the end-user for a constituent and then displays the Notepad UI to add a notepad record.

More samples are included in the RE7\Help\Samples directory on each workstation.

## Setting Defaults

This is a simple example to add default information to different record types. It demonstrates two important ideas. First, the Data Object macro is available from several different data entry forms, so it is important to check the data object type before using any of its properties. Second, you have access to the data object and all its child collections and classes, so you can add, edit, and delete information when necessary.

```
Public Sub SetDefaults(oRecord As IBBDDataObject)

    'Check the record type
    If TypeOf oRecord Is CRecord Then

        Dim oConstit As CRecord
        Dim oPhone As CConstitAddressPhone

        'Setting the oRecord = oConstit is not necessary
        ' but it allows Intelli-sense to function for the object
        Set oConstit = oRecord

        With oConstit
            'Add a phone to the preferred address
            Set oPhone = .PreferredAddress.Phones.Add

            'Set the phone type
            If .Fields(RECORDS_fld_KEY_INDICATOR) = bbki_INDIVIDUAL Then
                oPhone.Fields(CONSTIT_ADDRESS_PHONES_fld_PHONETYPE) = "Home"
            Else
                oPhone.Fields(CONSTIT_ADDRESS_PHONES_fld_PHONETYPE) = "Business"
            End If

            'Clean up object references
            Set oPhone = Nothing
        End With

    ElseIf TypeOf oRec Is CGift Then
        'Add default gift information

    End If

End Sub
```

## Adding Notepad Records

This sample uses many VBA features. It is a standard macro that prompts the end-user for a constituent and then displays the Notepad UI to add a notepad record.

```
Attribute VB_Name = "NewNote"
Option Explicit

Public Sub Add_Notepad()

    Dim lID As Long 'Use this variable to store the ID of the constituent.

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

    Dim oQuickSearch As IBBMiscUI
    'Search for the constituent using the Quick Search form
    Set oQuickSearch = ReService.CreateServiceObject(bbsoMiscUI)
    oQuickSearch.Init REApplication.SessionContext

    lID = oQuickSearch.PromptForDataObject(SEARCH_CONSTITUENT, _
        "Constituent", "Search for a Constituent; Enter Surname first")

    'If a constituent is found, then create a new notepad for the constituent and open
    'the new notepad form.
    If lID > 0 Then
        Dim oRecord As CRecord
        Set oRecord = New CRecord
        oRecord.Init REApplication.SessionContext

        Dim oNotepadForm As CNotepadForm
        Set oNotepadForm = New CNotepadForm
        oNotepadForm.Init REApplication.SessionContext

        oRecord.Load lID

        With oNotepadForm
            Set .NotepadObjects = oRecord.Notepads
            .FormCaption = oRecord.Fields(RECORDS_fld_FULL_NAME)
            .ShowForm Nothing, , True
        End With
    End If
End Sub
```

```
oRecord.Save
    oRecord.CloseDown
    Set oRecord = Nothing

    oNotepadForm.CloseDown
    Set oNotepadForm = Nothing
Else
    MsgBox "Constituent not found."
End If

oQuickSearch.CloseDown
Set oQuickSearch = Nothing

ReService.CloseDown
Set ReService = Nothing

End Sub
```

## The Active Objects

### REApplication Object

The RE\_Application Object offers two events: UIOpening and UIClosing.

### The Active Data Objects

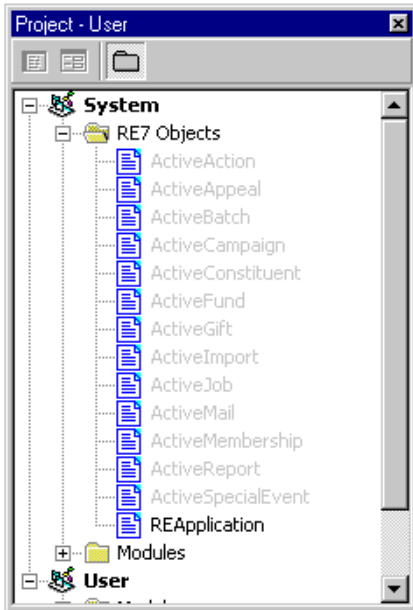
Active Data Objects are the most common objects. The name begins with the program that contains that object. For example, the ActiveAction object pertains to action records.

### Active Process Objects

This section introduces the active objects involved with processes in *The Raiser's Edge*. **Import** and *Reports* are two examples.

## The REApplication Object

The REApplication object represents the overall program. It provides access to a valid SessionContext which is needed to initialise any *Raiser's Edge* object. For more information on SessionContext and how to access it from VBA, see the Essentials chapter. The REApplication object also provides two events for which you can write code to execute. The “UIOpening Event” on page 112 fires as the shell opens and the “UIClosing Event” on page 112 fires as the shell is closed. The REApplication object also has one read only property, Version, which returns the complete version, including the build number, of *The Raiser's Edge*.



### UIOpening Event

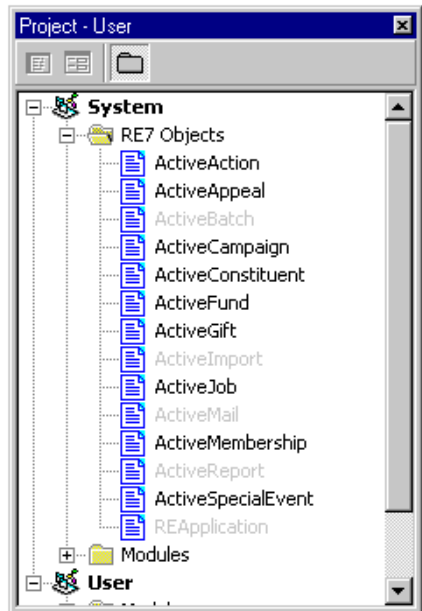
Using the UIOpening event, you can execute a section of code whenever an end-user begins a *Raiser's Edge* session. For instance, you can present your end-user with a reminder as she begins or connect to another application's database to transfer data between applications. The UIOpening event fires immediately after the shell appears on the screen, but before the Home page displays.

### UIClosing Event

Using the UIClosing event, you can execute a section of code just before *The Raiser's Edge* completely closes. It gives you the chance to close down a connection to another application's database or to check and make sure the end-user has correctly completed a specific task. With the UIClosing event, you can also stop the shell from closing by setting the bCancel variable to True. This stops the program from closing and returns the end-user to *The Raiser's Edge* shell.

## The Active Data Objects

Active data objects are provided for each top-level object. When creating macros, you place code in the events provided with these objects. First, run the Raiser's Edge Code Wizard to select the event and allow some code to be generated for you. For more information about the Raiser's Edge Code Wizard, see "Active Object Macros and The Raiser's Edge Code Wizard" on page 102.



## Active Data Object Events

Any code written for these events is executed for every end-user whenever one of these events fires. You need to log into *The Raiser's Edge* as Supervisor to view or edit this code. With these events you can:

- Notify end-users of required fields, and prevent them from saving records that do not have data in these fields.
- Prevent end-users from deleting records meeting a certain criteria. For example, constituents who have recently given a gift.
- Maintain a separate database with information entered into or generated by *The Raiser's Edge*.
- Send an email to specific individuals at your organisation when certain conditions are met. For example, you could notify the Director of Development whenever a major contribution is made.

The following examples use these events to maintain a separate database of Major Donors and their total contributions.

- "The BeforeOpen Event" on page 114—Occurs before a top level object is opened
- "The BeforeSave Event" on page 115—Occurs before a top level object is saved
- "The AfterSave Event" on page 116—Occurs after a top level object is saved
- "The CloseRecord Event" on page 119—Occurs when a top level object is closed
- "The BeforeDelete Event" on page 119—Occurs before a top level object is deleted

- “The AfterDelete Event” on page 120—Occurs after a top level object is deleted

### Please remember....

The example code provided is abbreviated and does not contain all the necessary error trapping. You need to add a reference to the Microsoft DAO Object Library to access the database objects, and a reference to the Microsoft Outlook Object Library to access the objects used in the email example.

## The BeforeOpen Event

The BeforeOpen event fires before the UI form for a new or existing record displays. This gives you the opportunity to perform any pre-processing to the underlying data object, or to display instructions for your end-users.

The event passes you oRecord, an object representing the active object. The oRecord object is already initialised, so there is no need to call the Init method or the CloseDown method at the end of your subroutine. Using the oRecord object, you can set properties such as Fields or use the objects methods before the form appears to your end-user.

Place the code below in the BeforeOpen event of the ActiveConstituent object. It notifies end-users that they are editing a Major Donor.

```
Private Sub RaisersEdgeRecord_BeforeOpen(oRecord As Object)

    Dim oMDB As Database
    Dim oMajorDonor As Recordset

    Set oMDB = OpenDatabase("d:\MajorDonors.mdb")
    Set oMajorDonor = oMDB.OpenRecordset("Constituents")

    With oMajorDonor
        .Index = "PrimaryKey"
        .Seek "=", oRecord.Fields(RECORD_fld_ID)

        If Not .NoMatch Then
            sMsg = "This constituent is a Major Donor."
            sMsg = sMsg & "The Director of Development should be _
                notified of any significant changes to this record."

            MsgBox sMsg
        End If
    End With

    oMajorDonor.Close
    Set oMajorDonor = Nothing

    oMDB.Close
    Set oMDB = Nothing

End Sub
```



## The BeforeSave Event

The BeforeSave event fires before any Save operation is performed (**Save**, **Save and New**, **Save and Close**). With this, you can enforce any Business Rule specific to your organisation.

The event passes an *oRecord* that represents the active object and bCancel that, if set to True, cancels the save operation and returns focus to the active form. The oRecord object is already initialised, so there is no need to call the Init method or the CloseDown method at the end of your subroutine.

After the code is complete, *The Raiser's Edge* still performs the usual validation to guarantee all required fields are completed and date fields contain valid dates.

This code sample determines if a donation qualifies the constituent as a Major Donor and determines if the Director of Development needs to be notified. The code below is placed in the BeforeSave() event of the ActiveGift object.

```
Private Sub Gift_BeforeSave(oRecord As Object, bcancel As Boolean)

    'Before we save the gift record,
    '    1) Determine if this will make the person a Major Donor
    '        by checking the amount given
    '    2) See if we need to mark the constituent record with
    '        the Constituent Code, Major Donor
    '    3) Check the MajorDonor.Mdb to see if the Director of
    '        Development needs to be notified
    '
    '    The module level variables, mbAddMajorDonorCC and mbNotifyDOD
    '        will be used in the After_Save event

    Dim bHasMajorDonorCC As Boolean
    Dim oGift As CGift
    Dim oConstituentCode As CConstituentCode

    'This is not necessary, but allows you to use the Intellisense
    '    feature, which you don't get with the late bound oRecord.
    Set oGift = oRecord

    If oGift.Fields(GIFT_fld_Amount) > 1000 Then
        'Look through the person's constituent codes to see if they are
        '    marked as a major donor
        For Each oConstituentCode In oGift.Constituent.ConstituentCodes
            If oConstituentCode.Fields(CONSTITUENT_CODE_fld_CODE) = _
                "Major Donor" Then
                bHasMajorDonorCC = True
            Exit For
        End If
    Next oConstituentCode
```

```
'We'll only want to add the Major Donor constituent
'   code if it's not already there
mbAddMajorDonorCC = (Not bHasMajorDonorCC)

'Check the MajorDonor database to see if this person is already there,
'   if they are not, then notify the DoD
Dim oMDB As Database
Dim oMajorDonor As Recordset

Set oMDB = OpenDatabase("d:\MajorDonors.mdb")
Set oMajorDonor = oMDB.OpenRecordset("Constituents")

With oMajorDonor
    .Index = "PrimaryKey"
    .Seek "=", 1DatabaseID
    '.NoMatch will = True if the record is not found
    If .NoMatch Then mbNotifyDOD = True
End With

'Clean up
oMajorDonor.Close
Set oMajorDonor = Nothing

oMDB.Close
Set oMDB = Nothing

Else
    mbNotifyDOD = False
    mbAddMajorDonorCC = False
End If

Set oGift = Nothing
Set oConstituentCode = Nothing

End Sub
```

## The AfterSave Event

The AfterSave event fires after the data has been written to the database.

The event has one parameter, oRecord, representing the active data object; in this case, the oRecord is of type CGift.

In this code sample, we use the AfterSave event to notify the Director of Development of any gifts over £1000.00. This code also updates MajorDonors.Mdb and adds the Major Donor constituent code to the constituent record. The code below is placed in the AfterSave() event of the ActiveGift object.

```
Public Sub Gift_AfterSave(oRecord As Object)

    'All gifts over £1,000 will designate a constituent as a Major Donor.  If
    '   the person is a Major Donor, then
    '   1) We need to add the donor to our MajorDonors.Mdb
    '   2) If it's a new major donor, we need to notify the
    '       Director of Development
    '   3) We need to add the Constituent Code of Major Donor if
    '       it's not already there

    'The module level variable mbNotifyDOD was set in the Before_Save Event
    '   Since we only notify the DoD for new Major Donors, we know we need to
    '   add the donor to the MajorDonors.Mdb

    Dim oGift As CGift
    Set oGift = oRecord

    If mbNotifyDOD then

        'Send the email message
        Dim oOutlook As Outlook.Application
        Set oOutlook = CreateObject("Outlook.Application")

        Dim oMailItem As MailItem
        Set oMailItem = oOutlook.CreateItem(olMailItem)

        With oMailItem
            .To = "Director.of.Development@YourOrganisation.co.uk"
            .Subject = "Major Gift Alert"
```

```

.Body = oGift.Constituent.Fields(RECORDS_fld_FULL_NAME) & " (" & _
        oGift.Constituent.Fields(RECORDS_fld_CONSTITUENT_ID) & _
        ") has given a gift of " & _
        Format$(oGift.Fields(GIFT_fld_Amount), "$###,##0.00") & _
        " to the " & _
        oGift.Fields(GIFT_fld_Fund) & ". Please call to thank them."

        .Send 'Sends the email without user intervention

End With

Set oMailItem = Nothing
Set oOutlook = Nothing

'Update the MajorDonors.MDB
Dim oMDB As Database
Dim oMajorDonor As Recordset

Set oMDB = OpenDatabase("d:\MajorDonors.mdb")
Set oMajorDonor = oMDB.OpenRecordset("Constituents")

With oMajorDonor
    .AddNew
    .Fields("Donor_ID") = oGift.Constituent.Fields(RECORDS_fld_ID)
    .Fields("Donor_Name") = oGift.Constituent.Fields(RECORDS_fld_FULL_NAME)
    .Fields("Total_Amount") = oGift.Fields(GIFT_fld_Amount)
    .Update
End With

'Clean up
oMajorDonor.Close
Set oMajorDonor = Nothing

oMDB.Close
Set oMDB = Nothing
End If

```

```
'Add the constituent code.  
'The module level variable mbAddMajorDonorCC was set in the Before_Save Event  
If mbAddMajorDonorCC Then  
    Dim oConstituentCode As CConstituentCode  
    Set oConstituentCode = oGift.Constituent.ConstituentCodes.Add  
  
    oConstituentCode.Fields(CONSTITUENT_CODE_fld_CODE) = "Major Donor"  
    oGift.Constituent.Save  
End If  
  
Set oConstituentCode = Nothing  
Set oGift = Nothing  
  
End Sub
```

## The CloseRecord Event

The CloseRecord event fires just before the active data object and UI form closes. This gives you the opportunity to close down any objects you may have open when using this record.

In the MajorDonors example, we opened and closed the MajorDonors.Mdb whenever it was needed. You may want to open this database only once in the Before\_Open event, and then close it here.

## The BeforeDelete Event

The BeforeDelete event fires just before the active data object is to be deleted. This allows you to check any Business Rules specific to your organisation and cancel the delete process if needed.

The event passes oRecord that represents the active object and bCancel that, if set to True, cancels the delete operation and returns focus to the active form. The oRecord object is already initialised, so there is no need to call the Init method or the CloseDown method at the end of your subroutine.

In this example, we display a message indicating the constituent to be deleted is a recent donor. Place this code sample in the BeforeDelete event of the ActiveConstituent record.

```
Private Sub Constituent_BeforeDelete(oRecord As Object, bCancel As Boolean)

    Dim lResult As Long
    Dim oMDB As Database
    Dim oMajorDonor As Recordset

    Set oMDB = OpenDatabase("d:\MajorDonors.mdb")
    Set oMajorDonor = oMDB.OpenRecordset("Constituents")

    With oMajorDonor
        .Index = "PrimaryKey"
        .Seek "=", oRecord.Fields(RECORD_fld_ID)

        If Not .NoMatch Then
            lResult = MsgBox("This constituent is a major donor. _
                Are you sure you want to delete this record?", vbYesNo)
            If lResult = vbNo then bCancel = True
        End if
    End With

    oMajorDonor.Close
    Set oMajorDonor = Nothing

    oMDB.Close
    Set oMDB = Nothing

End Sub
```

## The AfterDelete Event

The AfterDelete event fires after a record has been deleted. The event passes both the Record ID and the Import ID from the record that was deleted. Both of these fields are unique and provide a way to identify the deleted record.

This example opens the Microsoft Access database called MajorDonors.Mdb and deletes a record if it exists. This code should be placed in the AfterDelete event of the ActiveConstituent object.

```
Private Sub RaisersEdgeRecord_AfterDelete(lDatabaseID As Long, sImportID As String)

    Dim oMDB As Database
    Dim oMajorDonor As Recordset

    Set oMDB = OpenDatabase("d:\MajorDonors.mdb")
    Set oMajorDonor = oMDB.OpenRecordset("Constituents")

    With oMajorDonor
        .Index = "PrimaryKey"
        .Seek "=", lDatabaseID

        If Not .NoMatch Then .Delete
    End With

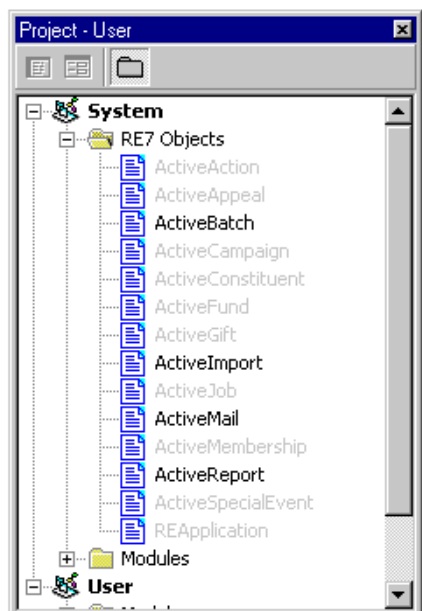
    oMajorDonor.Close
    Set oMajorDonor = Nothing

    oMDB.Close
    Set oMDB = Nothing

End Sub
```

## Active Process Objects

In addition to top level data objects, you can access and control certain processes using VBA. Through these service objects, you access *Batch*, *Mail*, *Reports*, and **Import**. The specific events provided vary based on the process. However, in general, you can act before and after the process and to handle exception processing.



## VBA Support in Batch

VBA in *Batch* provides four events so you can programatically access gift or constituent records before they are committed to the database. You can also create exceptions for constituents or gifts you do not want to commit to the database. You can specify the exception message so your end-users are able to determine what is wrong with the record. You can even make changes to a record that was previously an exception and instruct *Batch* to try again. The diagram below shows these events and the order in which they fire.

```
E1. BeforeCommit
E2. BeforeCommitRecord (If bCancel is set to True cancel the posting of this record and save
the custom exception message to be shown later on the exception report)
If there was an exception or bCancel was set to True in the BeforeCommitRecord then
E3. HandleException (If bRetry set to True, return to step E2)
E4. AfterCommit
```

In the following sections, we use the *Batch* events to maintain a batch history database. It keeps track of each gift batch posted; including who posted it, the total gift amounts posted, and the total gift amounts that were exceptions.

### The BeforePost Event

The BeforePost event fires once, just before the process to commit the records to the database begins. This way, you can set up any processes you want in place for processing the batch. For example, if you are transferring data from the batch to an external database, you can use this event to connect to that database.



The event passes `sBatchName`. This is a string that represents the batch number of the batch in progress. It also passes `lBatchType`, from the enum `bbVBABatchTypes`, which tells you if this is a constituent, gift, or time sheet batch. The last parameter is `bCancel` that, if set to `True`, discontinues the entire committing process.

```
Private Sub BatchVBAREcord_BeforePost(ByVal sBatchName As String, _
                                     ByVal lBatchType As bbVBABatchTypes, _
                                     bCancel As Variant)

    On Error GoTo BatchVBAREcord_BeforePost_Error

    'Only keep history for Gift batches
    If lBatchType = bbBatchType_Gift Then
        'moDB is a module level Database variable
        ' (requires a reference to a Microsoft DAO Library)
        Set moDB = OpenDatabase("d:\BatchHistory.mdb")

        'mbMaintainHistory is a module level boolean variable
        mbMaintainHistory = True
    End If

Exit Sub

BatchVBAREcord_BeforePost_Error:
    'Don't let them run batch unless they can access the history file
    mbMaintainHistory = False

    MsgBox "Error opening BatchHistory.MDB. Please see your system administrator"

    'Setting bCancel to TRUE will cancel the post process
    bCancel = True

End Sub
```

## The BeforePostRecord Event

The `BeforePostRecord` event fires just before each row in the batch is committed to the database. This way, you can access the underlying data object after it has been created, but before it adds to the database. You can pass some of this information to a separate database or you can verify that specific Business Rules for your organisation have been met. If the rules are not met, you can stop this record from being committed to the database.

The event passes `sBatchName`. This is a string that represents the batch number of the batch in progress. It also passes `lBatchType`, from the enum `bbVBABatchTypes`, which tells you if this is a constituent, gift, or time sheet batch. The next parameter is `oDataObject`. This variant is either of type `CRecord`, `CGift`, or `CTimeSheet`, depending on the batch type. You can access the entire object model for that object type at this time. You can change information, use *VB* to make a calculation and add that information to the record, or check to make sure all the information is entered according to your organisation's Business Rules.

If you do not want this particular record to add to the database, you can set `bCancel` equal to `True`. The record does not add to the database and lists as an exception on the Exception Report. You can even set `sExceptionMessage` equal to the reason for the exception so it prints on the Exception Report and shows your end-user what is wrong with the record. Any time a record is flagged as an exception, "The `HandleException` Event" on page 124 fires.

To continue with the previous example, here we use the BeforePostRecord event to total the dollar amount of cash gifts posted and other gifts posted.

```
Private Sub Batch_BeforePostRecord(ByVal sBatchName As String, _
                                   ByVal lBatchType As bbVBABatchTypes, _
                                   oDataObject As Variant, _
                                   bCancel As Variant, _
                                   sExceptionMessage As Variant)

    Dim oGift As CGift

    'mbMaintainHistory is a module level boolean variable
    If mbMaintainHistory Then
        'Since this is a Gift batch, oDataObject is returned as a CGift object
        ' Creating a CGift object is not necessary, but allows the Intelisense
        ' feature of VBA to recognise the object.
        Set oGift = oDataObject

        With oGift
            If .Fields(GIFT_fld_Type) = "Cash" Then
                mcurPostedCash = mcurPostedCash + .Fields(GIFT_fld_Amount)
            Else
                mcurPostedOther = mcurPostedOther + .Fields(GIFT_fld_Amount)
            End If
        End With

        Set oGift = Nothing

    End If

End Sub
```

## The HandleException Event

The HandleException event fires whenever a record in the batch is flagged as an exception. This way, you can change the record to correct the cause of the exception and try again. If you transferred information for each record to an external database in the BeforeCommitRecord event, you may want to remove that information using this event.

The event passes sBatchName. This is a string that represents the batch number of the batch in progress. It also passes lExceptionCode, from the enum bbVBABatchExceptionCodes, which tells you the reason for the exception. The next parameter is oDataObject. The variant is either the type CRecord, CGift, or CTimeSheet, depending on the batch type. You can access the entire object model for that object type at this time. The last parameter is bTryAgain. If set to True, *The Raiser's Edge* attempts to commit the updated record to the database again. It is important to make sure the cause of the exception has been corrected so there is no possibility of an end-user getting caught in a loop.

Using the HandleException event, you can total the dollar amount of cash gifts and other gifts that are exceptions.

```
Private Sub BatchVBAREcord_HandleException(ByVal sBatchName As String, _
                                           ByVal lExceptionCode As _
                                           bbVBABatchExceptionCodes, _
                                           oDataObject As Variant, _
                                           bTryAgain As Variant)

Dim oGift As CGift

Const DEFAULT_FUND = 99

-----

' mbMaintainHistory is a module level boolean variable
If mbMaintainHistory Then
    ' Since this is a Gift batch, oDataObject is returned as a CGift object
    '   Creating a CGift object is not necessary, but allows the Intelisense
    '   feature of VBA to recognize the object.
    Set oGift = oDataObject

    With oGift

        ' We can check the exception codes and try again if we can
        '   correct the problem
        If lExceptionCode = bbBex_FundNotFound AND _
            .Fields(GIFT_fld_Fund) <> DEFAULT_FUND Then
            ' In this case, we have a default fund set up to catch gifts
            '   where the fund was deleted between the time the batch
            '   was created and when it was posted

            ' Fix the fund
            .Fields(GIFT_fld_Fund) = DEFAULT_FUND

            ' Set bTryAgain and the system will again try to post the gift
            bTryAgain = True
        End With
    End With
End If
```

```
Else

    'Sum the exception amounts
    If .Fields(GIFT_fld_Type) = "Cash" Then
        mcurExceptionCash = mcurExceptionCash + _
                                .Fields(GIFT_fld_Amount)
    Else
        mcurExceptionOther = mcurExceptionOther + _
                                .Fields(GIFT_fld_Amount)
    End If
End If
End With
Set oGift = Nothing

End If

End Sub
```

### The AfterPost Event

The AfterPost event fires once after the entire committing process is complete. This way, you can “clean up” any objects or connections to other databases you use while the batch processes. You may also want to use this event to begin the acknowledgement process for gifts or to prepare a welcome letter for new constituents.

The event passes sBatchName. This is a string that represents the batch number of the batch in progress. The event also passes two longs; INumRecsPosted and INumExceptions. These represent the number of records that add successfully and the number of records that are exceptions, respectively.

To complete our example, you can add a row to the BatchHistory table (including the current date and batch name), the posted and exception totals from the BeforePostRecord and HandleException events, and the current User ID.

```
Private Sub BatchVBARecord_AfterPost(ByVal sBatchName As String, _
                                     ByVal lNumRecsPosted As Long, _
                                     ByVal lNumExceptions As Long)

    Dim oBatch As Recordset

    If mbMaintainHistory Then
        'Update the BatchHistory database with the details of the current batch
        With moDB
            Set oBatch = .OpenRecordset("BatchHistory")
            With oBatch
                .AddNew
                .Fields("RunDate") = Now()
                .Fields("BatchName") = sBatchName

                .Fields("PostedCash") = mcurPostedCash
                .Fields("PostedOther") = mcurPostedOther

                .Fields("ExceptionCash") = mcurExceptionCash
                .Fields("ExceptionOther") = mcurExceptionOther

                .Fields("UserID") = REApplication.SessionContext.CurrentUserID
                .Update
            End With
            oBatch.Close
            Set oBatch = Nothing
        End With
    End If
    moDB.Close
    Set moDB = Nothing

End Sub
```

## VBA Support in Import

VBA in **Import** provides four events so you can write code that executes during the importing process. You can also create exceptions, preventing unwanted records from entering the database. You can also specify the exception message so end-users can determine what is wrong with the record. In addition, you can make changes to a record that was previously an exception, and instruct **Import** to try again. The diagram below shows the events and the order in which they fire.

```
E1. BeforeImport
E2. BeforeImportRecord (If bCancel is set to True then cancel the save and save the custom
exception message to later be shown on the exception report)
If there was an exception or bCancel was set to True in the BeforeImportRecord then
E3. HandleException (If bRetry set to True, return to step E2)
E4. AfterImport
```

In the following example, we create two text files. One stores the name and national insurance number of imported records; the other accepts records that are exceptions.

### The BeforeImport Event

The BeforeImport event fires once, just before the importing of the records to the database begins. You can start any process you want to have in place while importing. For example, if you are also importing some of this data into a separate database, you can use this event to connect to that database.

The event passes sImportName. This is a string that represents the name of the import in progress. It also passes lImportType, from the enum bbVBAImportTypes, which tells you the type of the import, such as Constituent or Constituent Phone. The last parameter is bCancel which, if set to True, discontinues the importing process.

```
Public Sub ImportVBARecord_BeforeImport(ByVal sImportName As String, _
                                       ByVal lImportType As bbVBAImportTypes, _
                                       ByRef bcancel As Variant)

    'For constituent import files we need to provide a list of constituents
    ' imported and those that were exceptions. This routine opens the two text
    files
    ' that will hold then name and SSN of the records processed.

    If lImportType = bbImportType_Constit Then

        If MsgBox("Create import summary files?") = vbYes Then
            'mlExcFile and mlImpFile are module level variables
            ' that hold the two file handles

            mlExcFile = FreeFile
            Open "C:\Exc_" & sImportName & ".IMP" For Append As mlExcFile

            mlImpFile = FreeFile
            Open "C:\Imp_" & sImportName & ".IMP" For Append As mlImpFile
        End If

    End If

End Sub
```

### The BeforeImportRecord Event

The BeforeImportRecord event fires just before each row in an import file is imported into the database. You can access the underlying data object after it is created, but before it is added to the database. You may use this to send some information to a separate database. You can use it to verify that some specific Business Rules for your organisation have been met. If the specific rules have not been met, you can stop the record from being imported into the database.

The event passes sImportName. This is a string that represents the name of the import in progress. It also passes ImportType, from the enum bbVBAImportTypes, which tells you the type of the import, such as Constituent or Constituent Phone. The next parameter is oDataObject; the type of this variant depends on the import type. You can fully access the entire object model for that object type at this time. You can change information, use VBA to make a calculation and add that information to the record, or check to make sure all the information is entered following your organisation's Business Rules.

If you do not want this particular record to add to the database, you can set `bCancel` equal to `True`. The record is not added to the database and lists as an exception on the Exception Report. You can even set `sExceptionMessage` equal to reason for the exception so it prints on the Exception Report and informs your end-user what is wrong with this record. Any time a record is flagged as an exception, the “The HandleException Event (Active Import)” on page 129 event fires.

Here, we write the Name and National Insurance Number of the imported record to a text file.

```
Public Sub ImportVBAREcord_BeforeImportRecord(ByVal sImportName As String,
                                             ByVal lImportType As bbVBAImportTypes, _
                                             oDataObject As Variant, _
                                             ByRef bCancel As Variant, _
                                             ByRef sExceptionMessage As Variant)

    Dim oRec as CRecord

    'mlImpFile is a module level variable; a file handle
    '    for the imported records file
    If mlImpFile > 0 Then
        Set oRec = oDataObject
        Print #mlImpFile, oRec.Fields(RECORDS_fld_FULL_NAME) & "," & _
            oRec.Fields(RECORDS_fld_SOCIAL_SECURITY_NO)

        Set oRec = Nothing
    End If

End Sub
```

### The HandleException Event (Active Import)

The `HandleException` event fires whenever a record in an import is flagged as an exception. You change the record to correct the cause of the exception and try again. If you transferred information for each record to an external database in the `BeforeImportRecord` event, you may want to remove that information using this event.

The event passes `sImportName`. This is a string that represents the name of the import in progress. It also passes `lExceptionCode`, from the enum `bbVBAImportExceptionCodes`, which tells you the reason for the exception. The next parameter is `oDataObject`, the data type of this variant depends on the type of import, such as `Constituent` or `Constituent Phone`. You can fully access the entire object model for that object type at this time. The last parameter is `bTryAgain`. If set to `True`, *The Raiser's Edge* attempts to import the updated record to the database again. It is important to make sure the cause of the exception has been fixed, so there is no possibility of your end-user getting caught in a loop.

Here, we write the Name and National Insurance Number of the exception record to a text file.

```
Public Sub ImportVBAREcord_HandleException(ByVal sImportName As String, _
                                           ByVal lExceptionCode As
                                           bbVBAImportExceptionCodes, _
                                           oDataObject As Variant,
                                           ByRef bTryAgain As Variant)

    Dim oRec As CRecord

    'mlExcFile is a module level variable; a file handle for the exception file
    If mlExcFile > 0 Then
        Set oRec = oDataObject
        Print #mlExcFile, oRec.Fields(RECORDS_fld_FULL_NAME) & "," & _
            oRec.Fields(RECORDS_fld_SOCIAL_SECURITY_NO)

        Set oRec = Nothing
    End If

End Sub
```

### The AfterImport Event

The AfterImport event fires once after the entire importing process is complete. You can clean up any objects or connections to other databases you use while the import processes. You may want to use this event to start a process to send a letter to new constituents who have been imported or send an email reminding the staff that phone numbers have been updated.

The event passes sImportName. This is a string that represents the name of the import in progress. The event also passes two longs; lNumRecsImported and lNumExceptions. These represent the number of records that add successfully and the number of records that are exceptions, respectively.



For this example, we close the text files and inform the end-user that the import files have been created.

```
Public Sub ImportVBAREcord_AfterImport(ByVal sImportName As String, _  
                                       ByVal lNumRecsImported As Long, _  
                                       ByVal lNumExceptions As Long)  
  
    If mExcFile > 0 Then  
        Close mExcFile  
        Close mImpFile  
  
        MsgBox "Import summary files created"  
    End If  
  
End Sub
```

## VBA Support in Reports and Mail

This section provides two events that fire before and after the actual processing of the report or mail function. You can prevent the report or mail function processing, and if you are exporting information, you can specify that the name of the export file appears so you can automatically graph the information in a spreadsheet.

### The BeforeProcess Event

*Reports* and *Mail* support the same VBA events. The BeforeProcess event fires just before the mail or report function begins to process. You can cancel the process before it begins. This is helpful if you want to implement a customised security system for these functions.

The BeforeProcess event passes four parameters. The first is `lReportType/lMailType`, a long that can be used with the enum `EReR_ReportTypes` to determine the type of report or mail function being run (such as the **Gift Entry Validation** report or **Labels** mail task). The next parameter is `sParamName`. This is a string that represents the name of the actual parameter file used for the report or mail function. Next is `lAction`, a long that, when used with the enum `EReR_ProcessOptions`, tells you if the user is printing, print previewing, or exporting the report or mail function. The last parameter is `bCancel` that, if set to `True`, cancels the entire process before it starts.

In this example, we allow end-users to run the **Gift Detail and Summary** and **Gift Entry Validation** reports only after 5:00 PM (when we are sure gift entry has been completed for the day).

```
Private Sub ReportsVBAREcord_BeforeProcess(ByVal lReportType As Long, _  
                                           ByVal sParamName As String, _  
                                           ByVal lAction As Long, _  
                                           bCancel As Boolean, _  
                                           ByVal Reserved As Variant)  
  
    Select Case lReportType  
  
        Case ReR_GiftDetailandSummaryReport, ReR_GiftEntryValidation  
  
            If Time() < "5:00 PM" Then  
                MsgBox "This report can only be run after 5:00 pm."  
                bcancel = True  
            End If  
  
        End If  
  
    End Sub
```

## The AfterProcess Event

*Reports* and *Mail* support the same VBA events. The AfterProcess event fires after the mail or report function finishes processing and displays its output. If the function does not have output and “No Records Meet Criteria” displays when processing is done, the AfterProcess event does not fire.

The AfterProcess event passes four parameters. The first is lReportType/lMailType, a long that can be used with the enum EReR\_ReportTypes to determine the type of report or mail function being run (such as the **Gift Entry Validation** report or **Labels** mail task). The next parameter is sParamName. This is a string that represents the name of the actual parameter file used for the report or mail function. Next is lAction, a long that, when used with the enum EReR\_ProcessOptions tells you if the end-user is printing, print previewing, or exporting the report or mail function. The last parameter is sExportFileName, which if the Action performed was exporting, contains the full path for the export file name.

In this example, after exporting Today's Reports, a copy is sent by email to the Director of Development using the Microsoft *Outlook* Object Library.

```
Private Sub ReportsVBAREcord_AfterProcess(ByVal lReportType As Long, _
                                           ByVal sParamName As String, _
                                           ByVal lAction As Long, _
                                           ByVal sExportInfo As String, _
                                           ByVal Reserved As Variant)

    On Error GoTo ehSendReport

    'You will need to set a reference to Microsoft Outlook 9.x Object Library
    Dim oOutlook As Outlook.Application 'This starts outlook.
    Dim oMailItem As MailItem

    'If the user is printing the 'Today's Reports' then
    If sParamName = "Today's Reports" Then

        If Len(sExportInfo) > 0 Then
            'Create the objects needed for email
            Set oOutlook = CreateObject("Outlook.Application")
            Set oMailItem = oOutlook.CreateItem(olMailItem)

            With oMailItem
                .To = "Development.Director@Blackbaud.co.uk"
                .Subject = sParamName
                .Attachments.Add (sExportInfo)
                .Send
            End With

            'Close all object references
            Set oMailItem = Nothing
            Set oOutlook = Nothing
        End If

    End If

    Exit Sub

ehSendReport:

    MsgBox Err.Description, vbOKOnly

End Sub
```

## VBA Support in Query and Export

You can write code that executes for each row in *Query* and *Export*. With these macros, you are able to detect the start of the process, as well as modify the results using the special *VBA* user fields provided.

## VBA Support in Query

*Query's* VBA support attaches a specially defined VBA macro to a query. When various operations are performed using the query (such as **Run**, **Refresh**, or **Export**), this special macro is called. By selecting **Tools, Run Macro** from the query menu bar, the end-user can run the macro associated with the active query without having to perform another operation in query.

### Please remember....

Due to the fact that query results are displayed “as needed” on the Results tab, for the macro to run completely through all query results, the end-user needs to scroll to the end of the query or run the macro by selecting **Tools, Run Macro** from the query menu bar.

Using VBA, you can add customised user fields to a query's output. This feature works in concert with creating a macro. On the Output tab, the end-user can select a special field, which applies to output only. This is called a VBA user field. You can write a macro to populate this field with data from VBA. The data shows up anywhere in *The Raiser's Edge* that query results are displayed and processed (such as *Query*, *Export*, and the search screen). Remember, it is important to note this field cannot be used as a filter.

An example of this feature is to add custom fields to *The Raiser's Edge* search screen to pull data from external databases. Also, when exporting data directly from *Query* to Crystal Decision's *Crystal Reports*, VBA provides the ability to add fields if you have performed calculations in VBA. This way, the data gets to *Crystal* in a useful format.

## Creating a Query Macro

To create a query macro, create a Public Sub in either the User\_Macros or System\_Macros module. This subroutine can accept only one parameter that is an IBBQueryRow object (for more information, see the **Programmer's Reference** in the help file). When the macro is run, either by running the query or accessing the macro directly from **Tools, Run Macro** on the query menu bar, it is called (Number of Rows in the query) two times. The first time it is called, it returns a .BOF = True (Begin Of File) condition. This allows you to perform any initialisation operations. The event is then called for each row in the query. It is important to remember nothing you do here affects membership in the query. That is determined by the criteria of the query. The macro is called the last time with a .EOF = True (End Of File) condition so you can perform any shutdown or cleanup operations. The following code sample is an example of this.

```
'This sample requires a reference to Excel Object Library.
Option Explicit

Private moExcel As Excel.Application
Private moWorksheet As Excel.Worksheet

Public Sub SendQueryResultsToExcel(oRow As IBBQueryRow)

If oRow.BOF Then
    'Opens Excel
    Set moExcel = CreateObject("Excel.Application")
    moExcel.Visible = True

    'Add a new worksheet
    moExcel.Workbooks.Add
    Set moWorksheet = moExcel.Worksheets(1) '.Add
```

```
'Fills the first row with the field names from the query
Dim lHeads As Long
For lHeads = 1 To oRow.FieldCount - 1
    moWorksheet.Cells(1, lHeads) = oRow.FieldName(lHeads)
Next lHeads

ElseIf oRow.EOF Then
    ' Post Process Some Results In Excel
    moWorksheet.Columns("A:E").EntireColumn.AutoFit

    'Clean up
    Set moWorksheet = Nothing
    Set moExcel = Nothing

Else
    ' Fill In The Details
    Dim l As Long
    For l = 1 To oRow.FieldCount - 1
        'Uses the data from the query to move to Excel
        moWorksheet.Cells(oRow.RowNum + 1, l) = oRow.Field(l)
    Next l

End If

End Sub
```

Refer to the `IBBQueryRow` (for more information, see the **Programmer's Reference** in the help file) object for additional properties you may find helpful when you are moving through the results of the query.

## VBA User Field (Query)

One *VBA* user field is included in the field list per query type. If this field is selected as an output field, it is a read/write field you can manipulate using the *VBA* macro. This is the only field in the query results the end-user has write access to during query processing. *Query* does not put anything in this field. To populate this field, a *VBA* macro must be used.

Remember the *VBA* user field lists in the **Available Fields** box only once per query type. However, it can be selected more than one time and follows standard naming conventions. This way, more than one user defined field can be specified per query.

## Code Sample

This macro illustrates an example of using `BOF` (Begin of File), `EOF` (End of File), and the *VBA* user field. The one parameter for the macro must be of type `IBBQueryRow` (for more information, see the **Programmer's Reference** in the help file). For a constituent query with three output fields selected (in the following order):

Constituent Information

1. Constituent Information, Name
2. Constituent Information, Age

### 3. VBA User Field

```
Public Sub UserQueryMacro(oQueryRow As IBBQueryRow)

    Const NAME_FIELD = 1
    Const AGE_FIELD = 2
    Const VBA_FIELD = 3

    If oQueryRow.BOF Then
        'This code will be processed once (at the beginning of
        ' the query result set).
        MsgBox "Begin processing"

    ElseIf oQueryRow.EOF Then
        'This code will be processed once (at the end of the
        ' query result set).
        MsgBox "End processing"

    Else
        'This code will be processed once per row
        If oQueryRow.Field(AGE_FIELD) > 55 Then
            'You can only write to a "VBA User field"
            oQueryRow.Field(VBA_FIELD) = "Senior citizen discount"
        End If

    End If

End Sub
```

#### ➤ When the query runs:

1. The message “Begin processing” with an **OK** button displays. When the end-user clicks **OK**, the query continues.
2. For each row, if the second query field (**Age**) is greater than 55, the third query field (*VBA user field*) is set to “Senior citizen discount”.
3. The message “End processing” with an **OK** button displays.

Remember when viewing query results on the Results tab, the EOF (End of File) does not fire unless the end-user scrolls to the end of the result set. EOF fires when the end-user reaches the end of the result set, or if the query is closed.

### VBA Support in Export

Like *Query*, *Export*’s VBA support attaches a specially designed VBA macro to an export. When the export is run, this macro is called. You can add customised user fields to your export using VBA. This is accomplished by attaching a macro to the export. On the Output tab of the export record, the end-user can select a special field that applies to output only. This is called a VBA user field. You can write a macro to populate this VBA user field with data from another database or perform a calculation that is contained in the export file. This feature can only be used with “flat” style exports. For more information about “flat” style exports, see the *Query & Export Guide*.

## Creating an Export Macro

To create an export macro, create a Public Sub in either the User\_Macros or System\_Macros module. This subroutine can only accept 1 parameter that is an IBBExportRow (for more information, see the **Programmer's Reference** in the help file) object. When the export is run, the macro is called (Number of Rows in the export) two times. The first time it is called, the macro returns a .BOF = True (Begin Of File) condition. This allows you to perform any initialisation operations. The event is then called for each row in the export. The macro is called the last time with a .EOF = True (End Of File) condition so you can perform any shutdown or cleanup operations. The following code sample is an example of this.

```
Option Explicit

Private moExcel As Excel.Application
Private moWorksheet As Excel.Worksheet
Private msLetters() As String
Private mlNumLetters() As Long
Private mlMaxLetters As Long

Public Sub Export_Macro(oRow As IBBExportRow)

    Dim lCount As Long

    If oRow.BOF Then
        'This clears the module-level variables so we
        '    can run this macro multiple times
        Erase msLetters()
        Erase msLetters()

        mlMaxLetters = 0

        'Opens Excel
        Set moExcel = CreateObject("Excel.Application")
        moExcel.Visible = True

        'Add a new worksheet
        moExcel.Workbooks.Add
        Set moWorksheet = moExcel.Worksheets(1)

        moWorksheet.Cells(1, 1) = "Letter"
        moWorksheet.Cells(1, 2) = "Count"
```

```

ElseIf oRow.EOF Then
    'Fill in Worksheet
    For lCount = 1 To mlMaxLetters
        moWorksheet.Cells(lCount + 1, 1) = msLetters(lCount)
        moWorksheet.Cells(lCount + 1, 2) = mlNumLetters(lCount)
    Next lCount

    moWorksheet.Cells(mlMaxLetters + 2, 2) = _
        "=SUM(B2:B" & mlMaxLetters + 1 & ")"
    moWorksheet.Columns("A:B").EntireColumn.AutoFit

    'Clean up
    Set moWorksheet = Nothing
    Set moExcel = Nothing

Else
    For lCount = 1 To mlMaxLetters
        If msLetters(lCount) = oRow.Field(2) Then
            'We found a match so we increment the counter for that letter
            'and exit the sub and go to the next record in the export.
            mlNumLetters(lCount) = mlNumLetters(lCount) + 1
            Exit Sub
        End If
    Next lCount

    'This means we didn't find a match
    mlMaxLetters = mlMaxLetters + 1

    ReDim Preserve msLetters(1 To mlMaxLetters)
    ReDim Preserve mlNumLetters(1 To mlMaxLetters)

    msLetters(mlMaxLetters) = oRow.Field(2)
    mlNumLetters(mlMaxLetters) = 1

End If

End Sub

```

Refer to the `IBBExportRow` object (for more information, see the **Programmer's Reference** in the help file) for additional properties you may find helpful when moving through the results of the export.

### VBA User Field (Export)

One *VBA* user field is included in the field list per group defined for the export type. If this field is selected as an output field, it is a read/write field you can manipulate using the *VBA* macro. This is the only field in the export file the end-user has write access to during export processing. Export does not put anything in this field. To populate this field, a *VBA* Macro must be used.



Remember the *VBA* user field can be selected more than one time per group and follows standard naming conventions. This way, more than one user defined field can be specified per group.

### Code Sample

The following macro shows an example of using BOF, EOF, and the *VBA* user field. Remember all properties of the *IBBExportRow* are defined in the **Programmer's Reference** in the help file. For an export with three output fields selected (in the following order):

#### Biographical

1. Constituent name
2. Age
3. *VBA* User Field

```
Public Sub UserExportMacro(oExportRow As IBBExportRow)

    Const NAME_FIELD = 1
    Const AGE_FIELD = 2
    Const VBA_FIELD = 3

    If oExportRow.BOF Then
        'This code will be processed once (at the beginning of the file).
        ' Here you would connect to an external database or run setup code
        MsgBox "Begin processing"

    ElseIf oExportRow.EOF Then
        'This code will be processed once (at the end of the file).
        ' Here you would disconnect/close external databases if needed or run
        ' end of processing code
        MsgBox "End processing"

    Else
        'This code will be processed once per row
        If oExportRow.Field(AGE_FIELD) > 55 Then
            'You can only write to a "VBA User field"
            oExportRow.Field(VBA_FIELD) = "Senior citizen discount"
        End If
    End If

End Sub
```

#### ➤ When the export runs:

1. The message “Begin processing” with an **OK** button displays. When the end-user clicks **OK**, the export continues.
2. For each row, if the second export field (**Age**) is greater than 55, the third export field (*VBA* user field) is set to “Senior citizen discount”.
3. The message “End processing” with an **OK** button displays.

# VBA DLL Macros

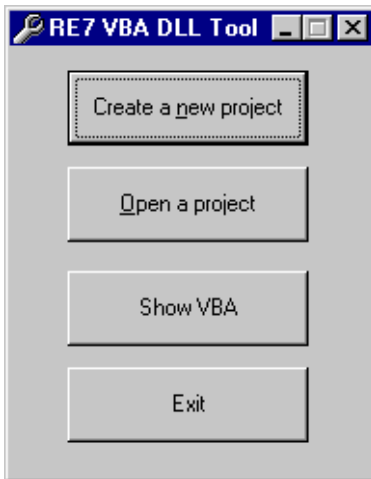
There are two main advantages to compiling your macros into a DLL (Dynamic Linking Library). The first is that you do not have to distribute the source code to your users. The second advantage is that the macros can be used from programs using *The Raiser's Edge* objects other than *The Raiser's Edge* shell. For example, you can build a custom *Raiser's Edge API* application that takes advantage of your macros.

## The DLL Tool

Any type of macro that can be created from within *The Raiser's Edge* shell can also be created using the DLL Tool. Refer to “Macros” on page 101 for details and code samples.

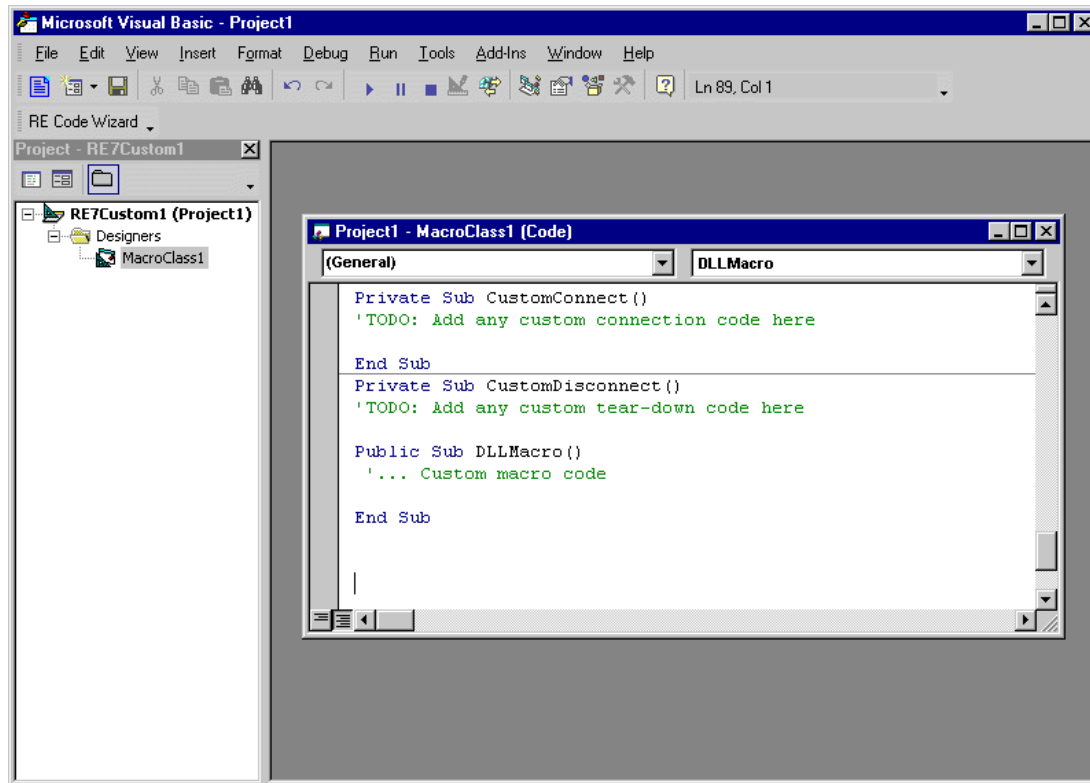
### Creating DLL Macros

First, enter the VBA IDE by running the RE7VBA.EXE program (located in *The Raiser's Edge* installation directory).



- **Create a new project**—Launches VBA and creates a new project. This is the easiest way to create RE7 DLLs.
- **Open a project**—Open an existing project for editing.
- **Show VBA**—Launches VBA without creating a new project.
- **Exit**—Closes the program.

After you click **Create a new project**, the VBA IDE loads with a project shell. The shell includes one module, **MacroClass1**, containing variable declarations and initialisation code. Do not make changes to this code. You can begin writing macros at the bottom of the **MacroClass1** module.



## Debugging DLL Macros

Debugging DLL macros is more complicated than debugging macros within the shell. To debug DLL macros, complete the following steps.

1. Make sure you log out of *The Raiser's Edge*.
2. From within the RE7VBA.EXE IDE, select the project (by default named "RE7Custom1").
3. Select **Tools, RE7Custom1 Properties**, from the menu bar.
4. On the Debugging tab of the Properties dialog, select the **Start Program**.
5. Select the option and type in the path to the re7.exe executable.
6. You are ready to debug. Set any breakpoints or debug messages you need.
7. Select **Run, Run Project** (Note: not **Run Macro**) from menu bar. This launches Re7.exe.
8. As Re7.exe calls your macro class you are able to step through your code in the VBA IDE.

Remember a DLL may become unregistered after debugging. If you stop debugging and your macros do not get called, you need to register the built DLL again, or rebuild it to automatically register it.

## Running DLL Macros

Before you can run DLL macros, the DLL needs to be compiled and properly registered. To compile the DLL, select **File, Make <name>.Dll** from the **VBA** menu bar. Once the DLL is compiled, the Active Object and Data Object macros are accessed the same way they are within the program shell. Standard macros are accessed from the **Tools, Run DLL Macro VBA** menu bar.

After compiling and registering the macro DLL, the macros are available the next time you log into *The Raiser's Edge*.

# Customising RE:Anywhere for Remote Access Using VBA

*RE:Anywher for Remote Access* does not support the integrated editing of VBA macros, as *The Raiser's Edge* user interface (UI) does. Any macros you create for *RE:Anywhere* must be created with the *RE:Anywhere* VBA DLL Tool. To access this, double-click REWeb7VBA.exe in your *The Raiser's Edge* install folder. Using this tool in *RE:Anywhere* is slightly different than using similar tools in *The Raiser's Edge*. In *The Raiser's Edge*, code is placed in the events of Active objects (for example, ActiveConstituent\_BeforeSave). In *RE:Anywhere*, IBBWebActive interfaces are implemented, and code is placed in their methods (for example, IBBWebActiveConstituent\_BeforeSave). The macro class template that appears when **Create a new project** is clicked differs accordingly.

### Please note...

You can create as many VBA macro DLLs as you want. However, if several VBA DLLs have code for the same event, there is no way to call the order. We do not recommend having code for an event in one DLL which depends on code for the same event in another DLL. If you have questions, contact Customer Support.

Any DLLs you create must be placed on the Web server hosting *RE:Anywhere* and registered. Do not place them on client workstations. In addition, any libraries to which your DLL has a reference must also be installed on the Web server hosting *RE:Anywhere*.

You cannot create conventional UI elements within your *RE:Anywhere* VBA macros. You can manipulate Blackbaud data objects, as well as read data from and write it to other applications (for more information see "VBA DLL Macro Permissions and Memory" on page 144). However, you cannot create *Raiser's Edge* UI elements (such as CConstituentForm) or even call the MsgBox function. If you do, the *RE:Anywhere* application shuts down. In addition, you cannot add forms to your VBA project. There are a few ways you can return information about the progress of your macro to the *RE:Anywhere* UI. For more information, see the next section, IBBVBAContext.AppendMessage.

## The RE:Anywhere for Remote Access VBA Object Hierarchy

When writing code using the VBA DLL Tool, you have access to the following object hierarchy off of the *RE:Anywhere* application object.

REWebApplicationMT

- IBBSessionContext
- IBBWebContext
  - IBBVBAContext
    - IBBVBAError
      - Number
      - Description
      - Source
      - Clear
    - AppendMessage
    - GetMessage
    - ClearMessage

The *RE:Anywhere* application object implements the REWebApplicationMT interface, which exposes the following methods.

- SessionContext As IBBSessionContext: the SessionContext is your hook into *API*, and can be used to initialise other *API* objects. For more information, see the *API Essentials Guide*.
- WebContext As IBBWebContext: an interface that provides access to certain functionality specific to *RE:Anywhere*.

The IBBWebContext interface exposes the following methods.

- VBAContext As IBBVBAContext: an interface that provides access to certain functionality provided to help you customise *RE:Anywhere* using *VBA*.

The IBBVBAContext interface exposes the following properties and methods.

- AppendMessage(sMessage As String, [sDelimiter As String]): Using AppendMessage you can pass text to the Web context that displays when the current process is complete. The content of the message clears at the beginning of any *VBA*-enabled process and accumulates over calls made to AppendMessage during the process. When the process ends, the contents of the message display to the *RE:Anywhere* user in a message box. You can include in the message text basic HTML tags such as bold, italics, or a simple HTML table if you want. The message box sizes dynamically to accommodate the HTML it receives (however, only to a certain point).
- GetMessage As String: returns the current contents of the message.
- ClearMessage: clears the contents of the message.
- Error As IBBVBAError: information about any run-time error encountered during the execution of your *VBA* code. This is used by *RE:Anywhere* to handle run-time errors robustly.

The IBBVBAError interface exposes the following properties and methods.

- Number As Long: the error number encountered by VBA at run-time.
- Description As String: the description of the error encountered by VBA at run-time.
- Source As String: a string identifying the source of the error encountered by VBA at run-time.
- Clear: allows the contents of the VBA error object to be cleared.

**Please note...**

The IBBVBAError interface exists so that information about run-time errors can communicate to *RE:Anywhere*. It is most likely you are not going to use this in your VBA code.

## VBA DLL Macro Permissions and Memory

You may want to use VBA DLL macros to integrate other databases or applications with *RE:Anywhere*. If you do, remember that *RE:Anywhere* runs using the user context assigned to it in Internet Information Services (IIS), if Anonymous Access is enabled (or as the network user logged into the Web server if Anonymous Access is disabled). The user context employed by IIS, on behalf of *RE:Anywhere*, must have appropriate security rights to any local or network resources accessed or modified by your *RE:Anywhere* code. For example, your Web server is CLIENTWEB1 and you have Anonymous Access enabled. A developer writes a VBA DLL macro that reads data from and writes data to an Access database called Custom.mdb. This is located on another server called CLIENTDATA2. The anonymous user in IIS on CLIENTWEB1 needs to have rights to access resources over the network that connect the two machines. In addition, the anonymous user needs access to the CLIENTDATA2 server with read and write access to Custom.mdb. A good way to manage this is to use the BBREWeb70Group created when *RE:Anywhere* is installed. For more information, see the *RE:Anywhere for Remote Access Installation Guide*.

Any VBA DLL macro runs in process with the *RE:Anywhere* application on your Web server. This affects the server's performance overall, for all users. The more complex a macro is, the more memory it uses, affecting overall performance. Make sure you monitor your code's complexity or install additional server resources if necessary. In addition, any memory leak in VBA DLL macro code could cause periodic problems for the Web server that hosts *RE:Anywhere*. You should be extremely careful to release any resources obtained during the execution of your macros.

## Debugging the VBA DLL Macro and Run-Time Errors

To use the interactive debugger in VBA to debug *RE:Anywhere* VBA DLLs, you need to disable **Anonymous Access** and enable **Integrated Windows Authentication** for all *RE:Anywhere* Web applications. To do this:

- Open **My Computer**. Select **Manage** and then **Services and Applications** from the treeview.
- Select **Internet Information Services**. Right-click the **REWeb70** application and select **Properties**.
- Select the Direct Security tab and click **Edit**.
- In the **Anonymous access** and **Authentication access** frames, uncheck **Anonymous Access** and any other options that are checked.
- Finally, check **Integrated Windows Authentication** and then repeat this process for **REWebRepView** and **REWeb70Viewers**. After you complete this, right-click **Internet Information Services** and select **Restart IIS**.

To debug an *RE:Anywhere* VBA DLL macro, log on to the *RE:Anywhere* Web server with administrative rights. Then, open the project properties screen by selecting **Tools, [Project Name] Properties** from the menu bar. On the Debugging tab of that dialog, select **Start browser with URL** and enter the URL of the login page of *RE:Anywhere* (using the host machine's name to identify it in the URL). For example, if *RE:Anywhere* is hosted on a Web server named CLIENTWEB1, the URL used on the Debugging tab should be "http://CLIENTWEB1/REWeb70/login.asp." You should access the Web application locally. You do not want the execution to pass through any proxy server or firewall. Set breakpoints as necessary and select **Run, Start Project** from the menu bar to launch the debugger.

#### Please remember...

*RE:Anywhere* needs to be able to read the interfaces of a VBA DLL from its file in order to access the DLL in debug mode. This means that any time a new interface or public object is added to the project, the DLL must be recompiled before that interface or object can be debugged.

Debugging is interfered if you recompile a project to a new filename without changing the project name and class name and if you do not unregister the old DLL. For example, you are working on a VBA DLL macro project named "WebTest", the name of your macro class is "CWebTest1," and you compile it to C:\WebTest.dll. When you launch the debugger, *RE:Anywhere* looks up the ProgID WebTest.CWebTest1 in the registry to find out that it is stored in C:\WebTest.dll. If the project is rebuilt to C:\WebTestNew.dll without unregistering the old C:\WebTest.dll, *RE:Anywhere* most likely still finds the old C:\WebTest.dll (which is still associated with the WebTest.CWebTest1 ProgID in the registry). If your breakpoints are not being hit when debugging, this is probably the cause.

If a bug in your VBA code results in a run-time error, a message box displays with a description of the error identifying the VBA DLL that raised it. If the event that raised the error has a parameter allowing you to cancel the event (for example, BeforeSave), it is not cancelled. However, code in downstream events is not executed. For example, if a run-time error occurs in the IBBWebConstituent\_BeforeSave code, any change made to the constituent (either by the user or by the code up to the point of failure) saves to the database. However, if there is also code in IBBWebConstituent\_AfterSave, it does not execute. An exception to this is the code in IBBWebActiveBatch\_BeforePostRecord and IBBWebActiveBatch\_HandleException. If there is a run-time error in one of these two methods, their code executes up to the point where the error occurs (for as many times as there are records or exceptions in the batch). The cancel flag in IBBWebActiveBatch\_BeforePostRecord does not take effect even if set to True before the run-time error occurs. If a run-time error occurs in either of these methods, IBBWebActiveBatch\_AfterPost does not execute.

#### Please note...

Blackbaud strongly encourages you to write robust VBA code that handles errors correctly. Otherwise, memory leaks could result from unreleased resources. All code examples should include proper error handling. Errors can be reported back to the *RE:Anywhere* UI using the IBBVBAContext.AppendMessage method. If you have questions, contact Customer Support.

## Sharing Code Between RE:Anywhere for Remote Access and The Raiser's Edge

Since the implementation of VBA DLL macros differs between *The Raiser's Edge* and *RE:Anywhere*, you may think you have to write all your code twice—once in the Active objects of VBA and once in the IBBWebActive methods of the *RE:Anywhere* DLL. If you want to code the same business logic into *The Raiser's Edge* and *RE:Anywhere*, you can use any third-party COM-enabled development tool (for example, *Visual Basic*) to write a COM object that implements your business logic, and call into that object from both *The Raiser's Edge* and *RE:Anywhere* VBA DLL macros. For example, you want to add a particular Constituent Code to any constituent before a constituent is saved. First, you can use *Visual Basic* to create a class (for example, CMyBusinessLogic) with a public method that adds the constituent code. Remember, this code can be called from either *The Raiser's Edge* or *RE:Anywhere*. The following code is an example of this.

```
Public Sub DoConstituentBeforeSave(ByRef oRecord As CRecord, ByVal bFromREAnywhere As Boolean, ByRef bCancel As Boolean, Optional ByRef oWebContext As IBBWebContext)

    With oRecord.ConstituentCodes.Add()
        .Fields(CONSTITUENT_CODE_fld_CODE) = "My Code"
        .Fields(CONSTITUENT_CODE_fld_DATE_FROM) = Format(Now(), "mm/dd/yyyy")
    End With

    If bFromREAnywhere Then
        oWebContext.VBAContext.AppendMessage "BeforeSave: Constituent Code Added."
    Else
        MsgBox "Constituent Code Added.", vbOKOnly, "BeforeSave"
    End If
```



The project containing this class needs references to **Blackbaud Raisers Edge 7 Objects** and **Blackbaud RE:Anywhere VBA Interfaces**. The project is then compiled into a DLL, which is then placed on both *The Raiser's Edge* client workstations and on the Web server hosting *RE:Anywhere*. Then, when writing *The Raiser's Edge* and *RE:Anywhere* VBA DLL macros, each needs only a reference to the business logic DLL and the following code:

```
' in RE:
Public Sub ActiveConstituent(oRecord As Object, bCancel As Boolean)
    Dim oMyBusinessLogic As CMyBusinessLogic

    Set oMyBusinessLogic = New CMyBusinessLogic
    oMyBusinessLogic.DoConstituentBeforeSave oRecord, False, bCancel
    Set oMyBusinessLogic = Nothing
End Sub

' in RE:Anywhere:
Private Sub IBBWebActiveConstituent_BeforeSave(oRecord As Object, bCancel As Boolean)
    Dim oMyBusinessLogic As CMyBusinessLogic
    Dim oWC As IBBWebContext

    Set oWC = REWebApplication.WebContext
    Set oMyBusinessLogic = New CMyBusinessLogic
    oMyBusinessLogic.DoConstituentBeforeSave oRecord, True, bCancel, oWC
    Set oMyBusinessLogic = Nothing
    Set oWC = Nothing
End Sub
```

## VBA Code Samples

For these examples, you need to import the actual code sample into *The Raiser's Edge*. Each sample contains a ReadMe.Txt file explaining the steps necessary to use the VBA macro. All VBA samples are located in the RE7\Help\Samples\VBA directory on each workstation.

### For more information...

Visit Blackbaud's Web site at [www.blackbaud.co.uk](http://www.blackbaud.co.uk) for software customisation 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 customising your *Raiser's Edge* software. You can also contact Customer Support for assistance.

Sample	Format	Description
Auto Word Merge	Standard Macro	Creates mail merge letters by interfacing with Microsoft Word.
Spell Check Notepad	Data Object Macro	Uses the spell check feature of Microsoft Word to spell check notepad objects.
Auto Add Constituent Codes	Visual Basic 6.0	When placed into the BeforeSave event of the ActiveConstituent object, this macro automatically adds "default" constituent codes if necessary.

New Notepad	<i>Visual Basic 6.0</i>	Using this macro, you can select a constituent and add a notepad record.
-------------	-------------------------	--

**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.co.uk](mailto:solutions@blackbaud.co.uk).

# Index

## A

- accessing, *see* opening
- active data objects 113
- active data objects events 113
- active object macro
  - creating 102
  - debugging 103
  - defined 101
  - running 103
- active objects 111
- active process objects 121
- active X
  - attributes grid 42
  - controls, defined 37
  - data grid 39
  - phones/email/links grid 45
- adding
  - child objects 30
  - data objects 20
  - notepad records 110
  - records using data object 20
- afterdelete event 120
- afterimport event 130
- afterpost event 126
- afterprocess event 132
- aftersave event 116
- annotation form 66
- API
  - library type 10
  - reference, set 11
  - session context 12
- API vs VBA 97
- attribute type server 63
- attributes grid, active X 42

## B

- Batch
  - afterpost event 126
  - beforepost event 122
  - beforepostrecord event 123
  - handleexception event 124
- beforedelete event 119
- beforeimport event 128
- beforeopen event 114
- beforepostrecord event 123
- beforeprocess event 131
- beforesave event 115

## C

- child objects
  - adding 30
  - collection types 22
  - defined 21
  - deleting 30
  - sorting collections 31
  - top collection 26
  - top view collection 28
  - view collection 28
- closedown 13
- clusercord event 119
- code sample
  - data objects 16
  - explanation 4
  - media form 72
  - query objects 47
  - report objects 55
  - table lookup handler 61
  - user interface objects 36
  - VBA 147
- code tables server 57
- code window 99
- code wizard 102
- collection
  - child object
    - child 22
    - programming 21
    - sorting 31
    - standard child 22
    - top 26
    - top view 28
    - view 28
  - data object, filtering 32
  - report object
    - categories 50
    - instances 53
    - types 52
- COM automation objects 5
- creating
  - active object macro 102
  - data object macro 106
  - DLL macros 140
  - Export macro 137
  - export macros 107
  - Query macro 134
  - query macros 107
  - standard macros 104
  - static queries 47
- Custom View* 85
- customising RE:Anywhere 142

## D

- data entry forms 35
- data grid active X 39

- data object macro
  - creating 106
  - debugging 107
  - defined 101, 105
  - running 107
- data objects
  - adding 20
  - code samples 16
  - database ID 15
  - defined 6, 13
  - deleting 20
  - filtering 32
  - hierarch 14
  - integrity 20
  - loading 15
  - top level 14
  - updating 18
  - validation 20
- database ID 15
- debugging
  - active object macro 103
  - DLL macros 141
  - export macros 107
  - query macros 107
  - RE:Anywhere 144
  - standard macro 104
- deleting
  - child objects 30
  - data objects 20
  - records using data object 21
- DLL macros 140
- documentation map 4

## E

- early bound objects 7
- errors 33
- Export
  - code sample 139
  - creating a macro 137
  - VBA support 136
  - VBA user field 138
- export macros
  - creating 107
  - debugging 107
  - defined 107
  - running 108

## F

- fields property 19
- filtering
  - data object collections 32
- forms designer 100

## H

- handleexception event 124, 129
- hierarchy, data object 14

## I

- IBBDataObject interface 81
- IBBMetaField interface 82
- Import
  - afterimport event 130
  - beforeimport event 128
  - beforeimportrecord event 128
  - handleexception event 129
- init 13
- initializing objects
  - closedown method 13
  - defined 13
  - init method 13
- integrity, data objects 20
- interfaces 5

## L

- library type
  - API 10
  - defined 7
  - early bound objects 7
  - VBA 8
- loading data objects 15

## M

- macro samples
  - adding notepad records 110
  - setting defaults 109
- macros 101
- Mail
  - afterprocess event 132
  - beforeprocess event 131
- media form
  - code sample 72
  - defined 70
- misc UI 78
- models, object 6

## N

- notepad form 68

## O

- objects

- active 111
  - data 113
  - process 121
  - REApplication 112
- child
  - adding 30
  - collection types 22
  - defined 21
  - deleting 30
  - sorting collection 31
  - top collection 26
  - top view 28
  - view collection 28
- collections 21
- data 13
  - defined 6
  - early bound 7
  - filtering 32
  - IBBDataObject interface 81
  - initializing 5, 13
  - models 5, 6
  - releasing 5, 13
  - service 5
  - session context 11
  - user interface 5
- report
  - annotation form 66
  - attribute type server 63
  - categories collection 50
  - code sample 55
  - code tables server 57
  - defined 50
  - instances collection 53
  - media form 70
  - misc UI 78
  - notepad form 68
  - property viewer 73
  - search screen 75
  - table lookup handler 60
  - types collection 52
- service
  - defined 45
  - query objects 46
- user interface
  - code sample 36
  - data entry forms 35
  - defined 34
  - showing standard forms 35
- opening
  - query objects 46
  - standard child collection 25
  - VBA 97

## P

- phones/email/links grid, active X 45
- processing query result set 46

- programming language 4
- project window 99
- property viewer 73

## Q

- Query
  - code sample 135
  - creating macro 134
  - user field 135
  - VBA support 134
- query and export macros 101
- query macros
  - creating 107
  - debugging 107
  - defined 107
  - running 108
- query objects
  - code sample 47
  - creating static queries 47
  - defined 46
  - opening 46
  - processing result set 46

## R

- Raiser's Edge, The*
  - code wizard 102
  - search screen 16
- RE:Anywhere
  - customising 142
  - debugging DLL macro 144
  - DLL macro permissions 144
  - memory 144
  - object hierarchy 143
  - run-time errors 144
  - sharing code with Raiser's Edge 146
- REApplication object 112
- record
  - adding using data objects 20
  - deleting using data objects 21
- references
  - API 11
  - VBA 9
- releasing objects
  - closedown 13
  - defined 13
  - init 13
- report objects
  - annotation form 66
  - attribute type server 63
  - categories collection 50
  - code sampe 55
  - code tables server 57
  - defined 50
  - instance collection 53

- media form 70
- misc UI 78
- notepad form 68
- property viewer 73
- search screen 75
- table lookup handler 60
- type collection 52

- Reports
  - afterprocess event 132
  - beforeprocess event 131

- running
  - active object macro 103
  - data object macro 107
  - DLL macros 142
  - export macros 108
  - query macros 108
  - standard macro 105

## S

- sample code, *see* code sample
- search screen 75
- service objects 5
  - defined 45
  - query 46
- session context
  - API 12
  - defined 11
  - VBA 12
- setting defaults, macro samples 109
- sorting, collections 31
- standard child collection
  - accessing 25
  - defined 22
  - navigating 23
  - structure 23
  - updating 24
- standard macro
  - debugging 104
  - defined 101
  - running 105

## T

- table lookup handler
  - code samples 61
  - defined 60
- top level data objects 14
- transactions 5
- translations 84
- type library 5
- type, library 7

## U

- UI closing event 112
- UI opening event 112
- UI, *see* user interface
- updating
  - data objects 18
  - standard child collection 24
- user field 135
- user interface objects
  - code sample 36
  - data entry forms 35
  - defined 34
  - showing standard form 35
- using documentation 4

## V

- validation, data objects 20
- VBA
  - code samples 147
  - code window 99
  - defined 96
  - forms designer 100
  - macros 101
  - opening 97
  - project window 99
  - properties window 99
  - reference, set 9
  - session context 12
  - type library 8
- VBA in Batch 122
- VBA vs API 97
- Visual Basic for Applications, *see also* VBA 97