Automating Evolver with VBA

The purpose of this document is to introduce Evolver's Excel Developer Kit (XDK) and explain how you can use VBA to automate Evolver. The term "automate" simply means that you write a program in VBA to make Evolver do what you normally do through the Evolver user interface. For example, suppose you have an Evolver model with a budget constraint, and you would like to run Evolver repeatedly, each time with a different value for the budget. In this way, you could create an "efficient front" of optimal solutions for different budgets. This requires a considerable number of steps—the same steps—each time you do it. Wouldn't it be nice to click a button and have all of these steps magically happen? This is exactly the type of thing you will learn how to do with the Evolver XDK. In fact, it is surprisingly easy.

Introduction to VBA[1]

Visual Basic for Applications, or VBA, is the programming language for Excel. It is also the programming language for other applications in Microsoft Office: Word, PowerPoint, and Outlook, for example. It is even the programming language for some non-Microsoft products. To understand how this works, it helps to separate the name VBA into two parts: "Visual Basic" and "for Applications." You can think of Visual Basic, or VB, as the "backbone" programming language. It contains programming elements that all programmers use, regardless of the programming language or the type of development task. For example, it contains subroutines, variable types, logical constructs such as If-Then-Else, loops for repetitive tasks, arrays for storing lists, and much more. Actually, there are non-VBA versions of VB. For example, you might be familiar with VB6 or VB.NET. All of these versions of VB, including VBA, are extremely popular because VB is fairly easy to learn and quite powerful. In any case, to learn VBA for Excel or Evolver, you must first learn the backbone VB language.

The following is a typical segment of VB code. Even if you know nothing about programming, you will probably have little trouble understanding what this code accomplishes. Again, this is the attraction of VB. It is relatively easy to learn and understand. Note the lines that begin with an apostrophe. These lines are called **comments**. They are ignored by VBA when the program runs, but they are very useful for documentation.

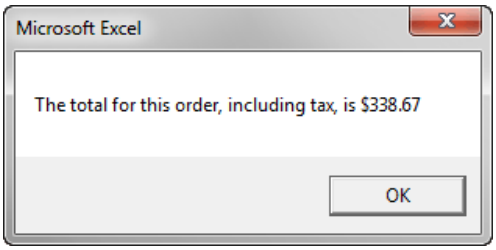
```
Sub Invoice()
    ' Declare variables.
    Dim nProducts As Integer, i As Integer
    Dim total As Currency, subTotal As Currency
    Dim nPurchased As Variant, unitPrice As Variant

    ' Define constants for this business.
    Const taxRate = 0.06
    Const cutoff1 = 50, cutoff2 = 100
    Const discount1 = 0.05, discount2 = 0.1

    ' Enter information about this order
    nProducts = 4
    nPurchased = Array(5, 2, 1, 6)
    unitPrice = Array(20, 10, 50, 30)

    total = 0
    ' Loop over all products purchased.
    For i = 1 To nProducts
        ' Calculate revenue, including possible discount, for this product.
```

[1] If you are already familiar with VBA for Excel, you can skip to the section "Introduction to the Evolver XDK" on page 5.

```
        subTotal = nPurchased(i) * unitPrice(i)
        If subTotal >= cutoff2 Then
            subTotal = (1 - discount2) * subTotal
        ElseIf subTotal >= cutoff1 Then
            subTotal = (1 - discount1) * subTotal
        End If
        ' Add to total for this order.
        total = total + subTotal
    Next
    ' Add tax.
    total = (1 + taxRate) * total
    ' Display result.
    MsgBox "The total for this order, including tax, is " & Format(total, "$#,#00.00")
End Sub
```

If you run this code in Excel, you will see the display (a message box) in Figure 1.

Figure 1 Result of VBA Program



Subroutines, Programs, and Macros

Before proceeding, it is useful to discuss three terms you often hear: subroutine, program, and macro. A **subroutine** is any section of code that begins with the keyword Sub and ends with the keyword End Sub. Its purpose is typically to perform one specific task. A **program** is a collection of one or more related subroutines that achieves an overall goal. In practice, long programs are often broken up into smaller subroutines for enhanced readability and easier debugging. A **macro** is essentially the same as a program, and it is the term favored by many Excel programmers. However, if your programs are relatively short, consisting of a single subroutine, the terms subroutine, program, and macro are practically synonymous and are often used interchangeably.

Object Models

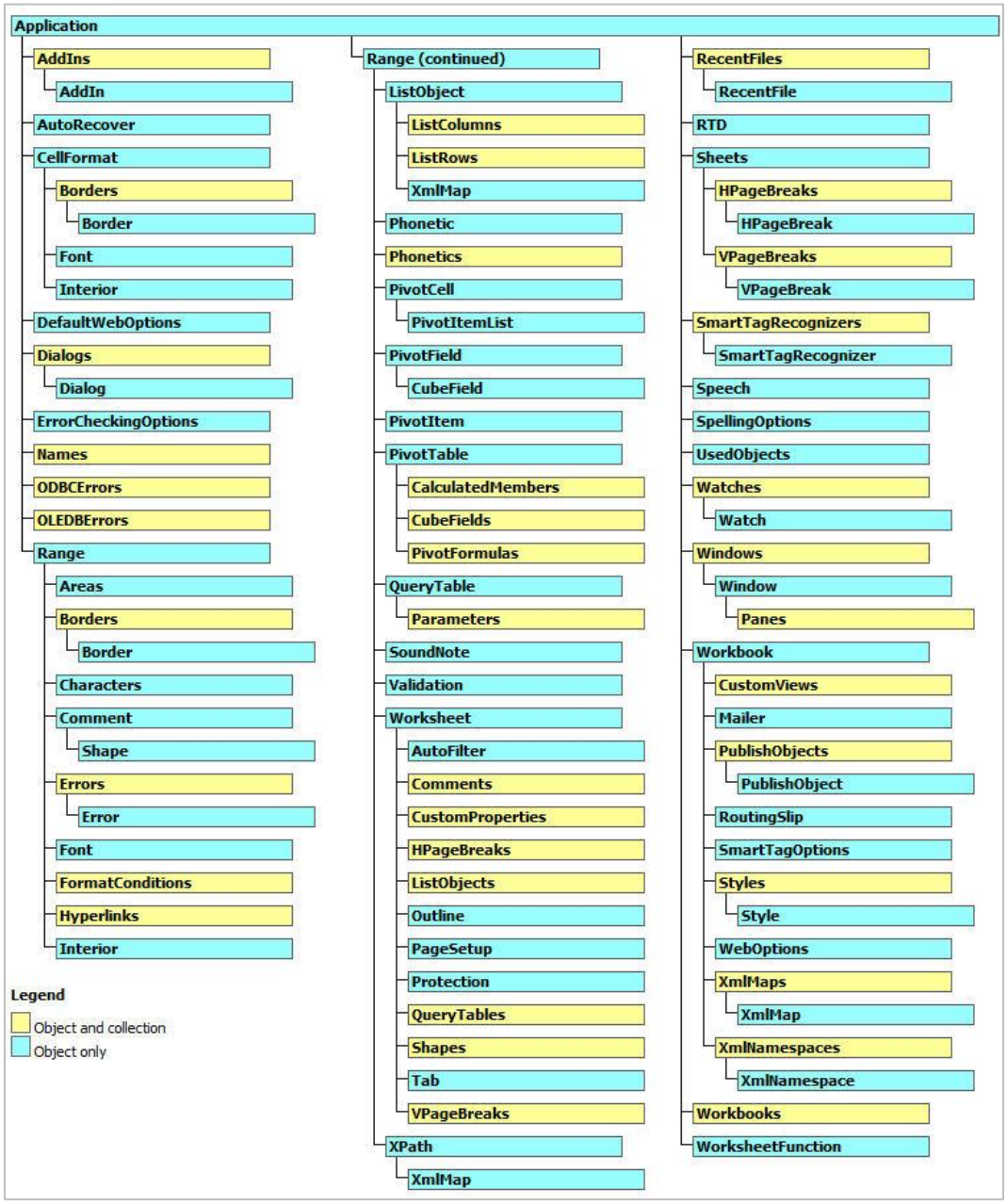
The above Invoice subroutine can be run in Excel, but it really has nothing to do with Excel. There are no references to ranges, formulas, worksheets, charts, or other items you use in Excel. The code is pure VB. But now we come to the “for Applications” part of VBA. All of the familiar items in Excel, which will now be called **objects**, are part of an **object model** that Excel provides to programmers through VBA. This means that you as a programmer can reference these objects in your VBA code. In short, this object model allows you to “make things happen” through VBA code, rather than through the usual Excel user interface. In this way, you can automate Excel with VBA.

So what does an object model, or Excel’s object model in particular, look like? At the top level, it is simply a list of things—objects—that the software contains. Excel’s object model is quite large because everything you see in Excel is an object. One of the most common objects is the Range object: any range of cells (including a single cell). But there are over a hundred more objects, including the Worksheet object, the Chart object, the PivotTable object, and so on. There are also singular and plural objects. For

example, there is the plural Worksheets object, the collection of all worksheets in an Excel workbook, and there is the singular Worksheet object, a particular worksheet in the workbook.

Figure 2 displays a partial list of the objects in Excel, plural (yellow) and singular (blue).

Figure 2 Excel Object Model



However, there is much more to an object model than the list of its objects. First, there is a **hierarchy** among many objects. Basically, this means that you often need to "drill down" to get to the object you want. A good example is a Range object, such as cell B3. The idea is that a range is contained in a worksheet, which is contained in a workbook. So the hierarchy is Workbook→Worksheet→Range, and the following line of code is typical:

```
ActiveWorkbook.Worksheets("Costs").Range("B3").Value = 10
```

You read this as: Starting in the active workbook, go to cell B3 of the worksheet named Costs, and enter the value 10. This concept of hierarchy is crucial to learning VBA for Excel or Evolver.

Second, most objects have **properties** and **methods**. (Methods are also called **functions**.) If you think of objects as nouns, then you can think of properties as adjectives and methods as verbs. Properties describe an object, and methods indicate what you can do to, or with, an object. For example, a Range object has a Value property, which was used in the above line of code. Similarly, a Font object has a Color property, a Size property, and many others. A typical object with methods is the Worksheet object. It has a Delete method, a Move method, and a few others.

Although less common, many objects also have **events** that they respond to. A good example is the Open event of a Workbook object. This event occurs—it "fires"—when you open the workbook. VBA allows you to write **event handlers** for the event. This is code that is executed when the event fires.

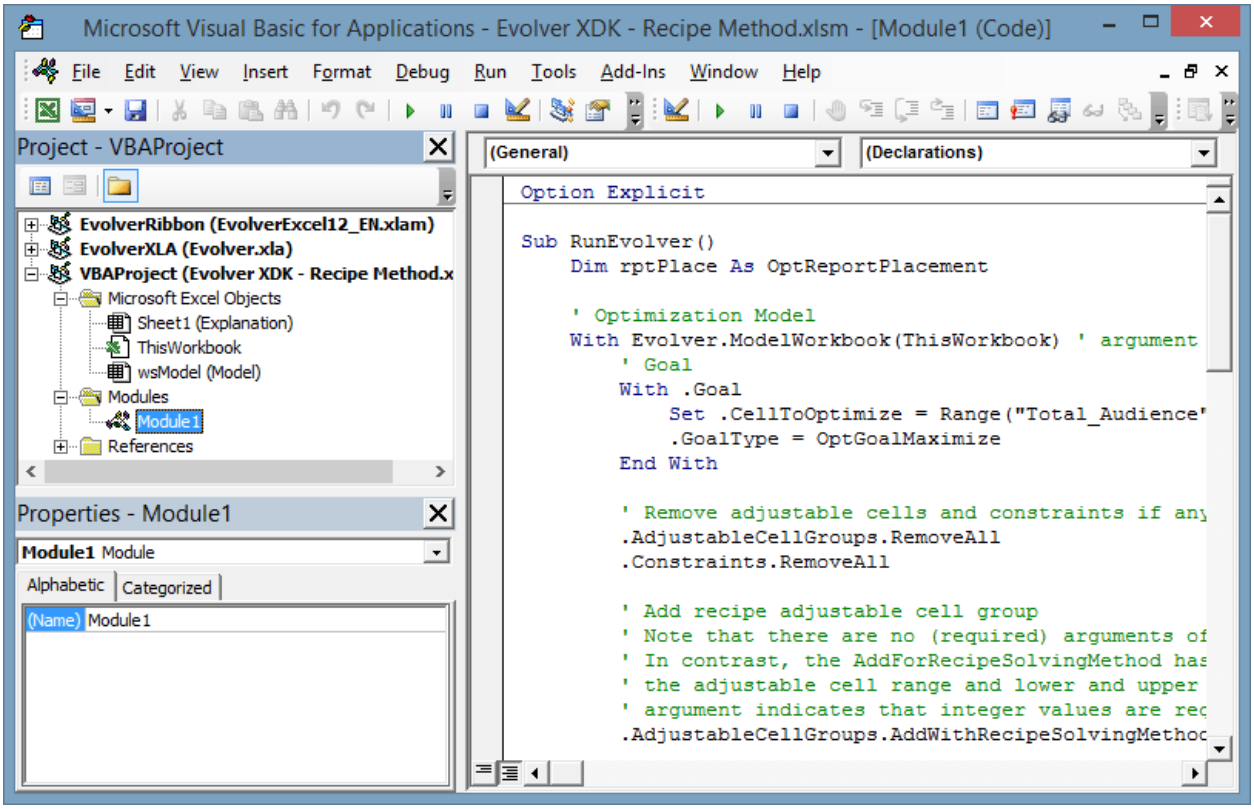
As you begin VBA programming for Excel, you gradually learn the most common objects, the hierarchies between them, their most common properties, methods, and events, and the syntax for using these in VBA code. It takes practice and perseverance, but if you want to make Excel "sing and dance" with the click of a button, this is time well spent.

By now, you should be starting to see the big VBA picture. Just as the Excel application can be automated through its object model, other applications can be automated through *their* object models. Microsoft Word is a typical example. Its objects are not the same as Excel's. Instead of Worksheet, Range, and other typical Excel objects, Word has Sentence, Paragraph, and other objects that are useful in word processing, and these objects have their own hierarchies, properties, methods, and events. Fortunately, if you already know VBA for Excel and you want to learn VBA for Word, all you need to learn is the object model for Word. The backbone VB language is exactly the same in both applications. Admittedly, it is not a trivial task to learn a new application's object model, but knowing VB for one application, like Excel, provides a big head start for learning VB for another application, like Word.

Using the Visual Basic Editor (VBE)

You will be doing all of your programming in the Visual Basic Editor (VBE). The easiest way is to open VBE from Excel is to press **Alt+F11**. Alternatively, if the Excel Developer tab is visible, you can click the **Visual Basic** button on its ribbon. The VBE window appears in Figure 3. In particular, the Project pane on the left shows a list of all open workbooks. (To get back to the Excel window, you can click the Excel icon on the toolbar below the menu bar.)

Figure 3 Visual Basic Editor



As you can see in the figure, the selected file contains a module. You will write all of your code in modules. (Actually, there are exceptions to this, but they aren't discussed here.) By default, a file doesn't contain a module, but you can insert one through the Insert menu. Then you will see a big white space on the right where you can start typing your code. Some of the code for this particular module appears in the figure.

Introduction to the Evolver XDK

Like Excel and Word, Evolver has its own object model, as shown in Figure 4. Assuming you are familiar with the Evolver add-in, there shouldn't be any surprises in this object model. It can be used to change application settings, designate the goal to optimize, the adjustable cells, and constraints, and run the optimization. The details of these possibilities are discussed in some detail later in this document and in the accompanying Evolver XDK example files.

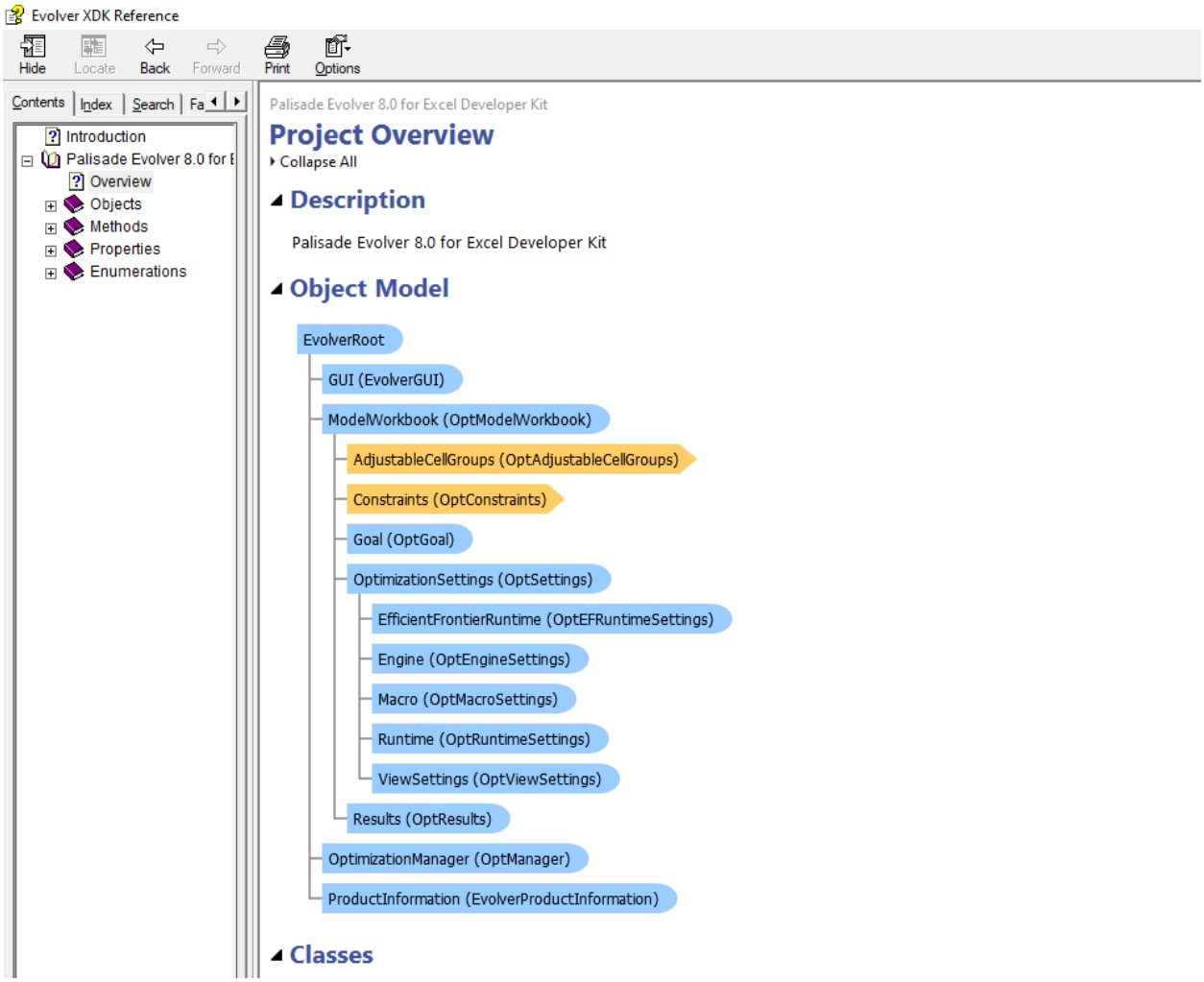
Two questions probably come to mind. Do you really need to learn how to automate Evolver with VBA, and, if you do, how do you learn the language?

There are at least two reasons why you might want to automate Evolver with VBA. First, it lets you automate tasks that you perform repeatedly. For example, suppose you need to download the most current data from a website, incorporate this new data into an existing Evolver model, modify adjustable cell ranges and constraints as necessary, and run the optimization. If you do this once or twice, it is probably not worth the effort to write a VBA program to automate the process. However, if you do this repeatedly, a VBA program can replace *many* mouse clicks with a *single* click. Indeed, this is why VBA for

Excel is so popular among users—it lets them automate repetitive tasks with the click of a button. This same motivation certainly applies to repetitive tasks in Evolver.

Second, suppose you are developing models for colleagues or clients who have little experience with Evolver. Instead of teaching them how the Evolver user interface works, it might be easier to develop a VBA application that allows them to simply click a button to make implement the entire optimization process.

Figure 4 Evolver Object Model



Assuming you think VBA for Evolver is for you, how do you learn the language? First, you need some facility in VBA for Excel. As discussed earlier, this requires you to learn the fundamentals of the VB "backbone" language and the basics of the Excel object model. There are a number of reference books on VBA for Excel, as well as Excel's own online help. One recommendation is the first 10 chapters of *VBA for Modelers* by Albright. This book provides concise explanations and plenty of example code to get you up to speed in both the VB language and the Excel object model.

Once you are familiar with VBA for Excel, you have at least four options—which are not mutually exclusive—for extending your knowledge to VBA for Evolver.

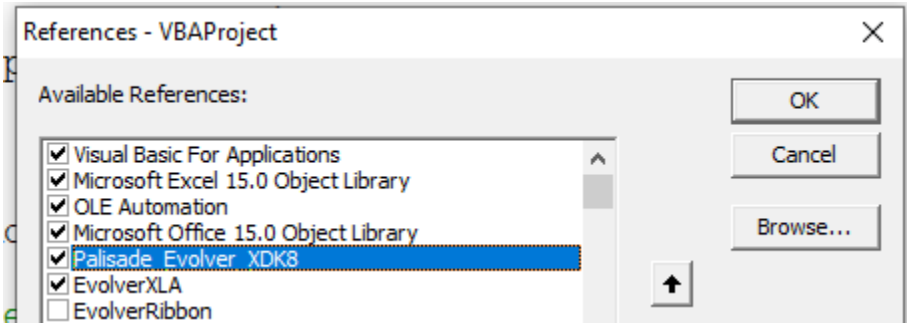
1. You can continue reading this document. It provides an overview of what can be done, and it provides sample code for how to do it.
2. You can do what all good programmers do—mimic code written by others. Examples of programs written for Evolver are provided in a number of Evolver XDK example files. (They can be found from the Automating Evolver (XDK) item in the Evolver Resources menu.) Once you see how something is done, such as designating adjustable cells or running an optimization, you will probably have little difficulty adapting the code for your own use.
3. You can open the Reference document from the Automating Evolver (XDK) item under the Evolver Resources menu and start browsing. This document is quite long, and there is no need to read it from beginning to end, but it gives you a quick sense of the Evolver object model, and it often enables you to find the specific details you need for your program.
4. You can open the Object Browser that is part of the Visual Basic Editor. This is particularly useful if you are in the midst of writing a program and need information on one specific detail. The Object Browser is discussed later in this document.

By the way, if you are familiar with VBA for Excel, you might want to start by *recording* macros for Evolver, just as you can do for Excel. Unfortunately, this doesn't work. If you turn on the recorder and start clicking Evolver buttons, you will get *some* recorded code, but it will be entirely irrelevant. To automate Evolver, *you* have to write the code.

Setting Library References

Before you can access the Evolver XDK with VBA code, you must first set references to two Evolver libraries (from References in the Tools menu of the Visual Basic Editor). These are the **EvolverXLA** and **Palisade_Evolver_XDK8** references, illustrated in Figure 5. (The former references the Evolver.xlam add-in file. The latter contains the Evolver Object Model; the reference to it can be added by browsing to the following file: C:\Program Files (x86)\Palisade\System\Palisade.Evolver.XDK8.tlb.) This should be the first step before writing any VBA macros to automate Evolver.

Figure 5 Evolver References



Evolver Automation Guidelines

This section provides some guidelines for automating Evolver with VBA code. The guidelines in this section are purposely kept brief and are intended only to give you the "lay of the land" and alert you to a few technical issues. To see complete applications of Evolver automation, you should look at the accompanying Evolver XDK example files. And finally, you can always visit the Evolver XDK documentation or the Object Browser in the Visual Basic Editor.

By the way, you will notice that all of the formal names of Evolver objects start with Opt, as in OptOptimizationSettings. However, there are typically referenced by a property or method that doesn't include the Opt prefix, such as OptimizationSettings.

Getting Started: The "Root" Object

You start with the "root" object **EvolverRoot**. Actually, you can simply type **Evolver** to reference this root object. For example, assuming you have set the references mentioned earlier, you can write Evolver.Optimize to run an optimization.

Technical Note: Evolver Function versus EvolverRoot Object

There is a subtle issue you should be aware of. As it is used here, **Evolver** is a *function* in the EvolverXLA library that returns a **EvolverRoot** *object* in the **Palisade_Evolver_XDK8** library. To put it another way, if you did *not* reference the EvolverXLA library and wrote a typical line such as the following, the Evolver reference wouldn't be recognized.

```
Evolver.ModelWorkbook().Goal.GoalType = OptGoalType_Maximize
```

However, if you *do* reference EvolverXLA, as you should, you can simply write Evolver, followed by any of its properties or methods, and it will work fine. (In particular, you do *not* need to declare Evolver as an object variable.)

Setting Up and Running Optimizations

The steps for setting up and running an optimization are straightforward, and they vary very little from one optimization model to another. You create an **OptModelWorkbook** object, and from it, you define the goal, the adjustable cells groups, the constraints, and the optimization settings. Then, after optimizing with the line **Evolver.Optimize**, you can report optimization results.

The following code, taken from the file **Evolver XDK – Recipe Method.xlsm**, is typical. Here are the basic steps.

1. It uses the outer With block to create the OptModelWorkbook object (implicitly).
2. It specifies that the goal is to maximize the value in the Total_Audience cell.
3. It removes any previous adjustable cell groups and constraints, just in case.
4. It adds an adjustable cell group of the recipe type. The range for this cell group is the Number_of_ads range, its lower and upper limits are in the Minimum_ads and Maximum_ads ranges, respectively, and the last argument (optional) indicates that the adjustable cells should contain integers.
5. It adds two hard constraints. The first is that the total cost should be less than the budget, and the second is that radio costs should not exceed TV costs.
6. It changes some optimization settings.
7. It runs the optimization.
8. It generates an optimization summary report and a log of the optimization progress.

```
Sub RunEvolver()
    Dim msg As String
    Dim rptPlace As OptReportPlacement

    ' Optimization Model
    With Evolver.ModelWorkbook(ThisWorkbook) ' argument optional, defaults to ActiveWorkbook
        ' Goal
```

```
        With .Goal
            Set .CellToOptimize = Range("Total_Audience")
            .GoalType = OptGoalType_Maximize
        End With

        ' Remove adjustable cells and constraints if any
        .AdjustableCellGroups.RemoveAll
        .Constraints.RemoveAll

        ' Add recipe adjustable cell group
        ' Note that there are no (required) arguments of the AddWithRecipeSolvingMethod.
        ' In contrast, the AddForRecipeSolvingMethod has three required arguments:
        ' the adjustable cell range and lower and upper limits. The last (optional)
        ' argument indicates that integer values are required.
        .AdjustableCellGroups.AddWithRecipeSolvingMethod _
            .AdjustableCellRanges.AddForRecipeSolvingMethod _
                Range("Number_of_Ads"), Range("Minimum_ads"), Range("Maximum_ads"), True

        ' Constraints
        .Constraints.AddHardInSimpleMode _
         0, OptConstraintOperator_LessOrEqual, Range("TC"), OptLessOrEqual, Range("BD"), "Budget"
        ' There is no upper bound in the following, so the argument after OptNone is irrelevant
        .Constraints.AddHardInSimpleMode _
         0, OptConstraintOperator_LessOrEqual, Range("TV_minus_radio"), OptNone, 0, "Costs"

        ' Optimization settings
        With .OptimizationSettings
            .Engine.OptimizationEngine = OptEngine_Automatic
            With .Runtime
                .ProgressStoppingCondition = True
                .ProgressMaxChangeIsPercent = True
                .ProgressMaxChange = 0.01
                .ProgressTrialCount = 20000
            End With
        End With

        ' Run optimization
        Evolver.Optimize

        ' Report selected results
        With .OptimizationResults
            wsModel.Range("J3").Value = (.FinishTime - .StartTime) * 24 * 60 * 60
            wsModel.Range("J4").Value = .NumTrials
            If .OriginalValueWithoutPenalties > 1000000 Then ' not feasible
                wsModel.Range("J5").Value = "N/A"
                wsModel.Range("J5").AddComment "Original solution doesn't satisfy constraints"
            Else
                wsModel.Range("J5").Value = .OriginalValueWithoutPenalties
            End If
            wsModel.Range("J6").Value = .BestValueWithoutPenalties
        End With

        ' Generate a report in the model workbook
        With .Results
            .GenerateOptimizationSummary True, ThisWorkbook
            .GenerateLog True, True, ThisWorkbook
        End With
    End With

    wsModel.Activate
    Range("A1").Select
End Sub
```

You can also use the Evolver XDK inside macros to obtain information about an optimization in progress and control it, even if the model is set up and the optimization is started through the graphical interface, not through VBA. For example, the following macro will stop the optimization if there is no progress after 100 trials. (To ensure that this macro is called during the optimization, you need to specify its name in the Macros tab of the Optimization Settings dialog).

```
Sub MyAfterStorageRoutine()
   With Evolver.ModelWorkbook(ThisWorkbook).OptimizationResults
      If .NumTrials >= 100 Then
         If .BestValueWithPenalties = .OriginalValueWithPenalties Then
            Evolver.OptimizationManager.StopOptimization
         End If
      End If
   End With
End Sub
```

Some General VBA Tips

This guide concludes with a few VBA tips that you should know regardless of whether you are automating Evolver or Excel.

File Format (xlsm)

If you save a workbook that includes VBA code, you must save it as a macro-enabled (.xlsm) file. This is true for any Excel file that contains VBA code; it is not specific to Evolver files. Then if you open this .xlsm file later on, you will be warned that it contains macros. Make sure you elect to enable the macros; otherwise the VBA macros won't work.

Running a Macro

If you develop a VBA program for nontechnical users, you probably won't want them to see your code, either for proprietary reasons or because it would be too intimidating. However, it is easy to create a simple user interface for running the program. To do so, activate a worksheet, insert a shape such as a rectangle, right-click it, select **Assign Macro**, and select your macro. You can also insert descriptive text, such as **Run Program,** in the shape. From then on, a user can simply click the shape to run the program.

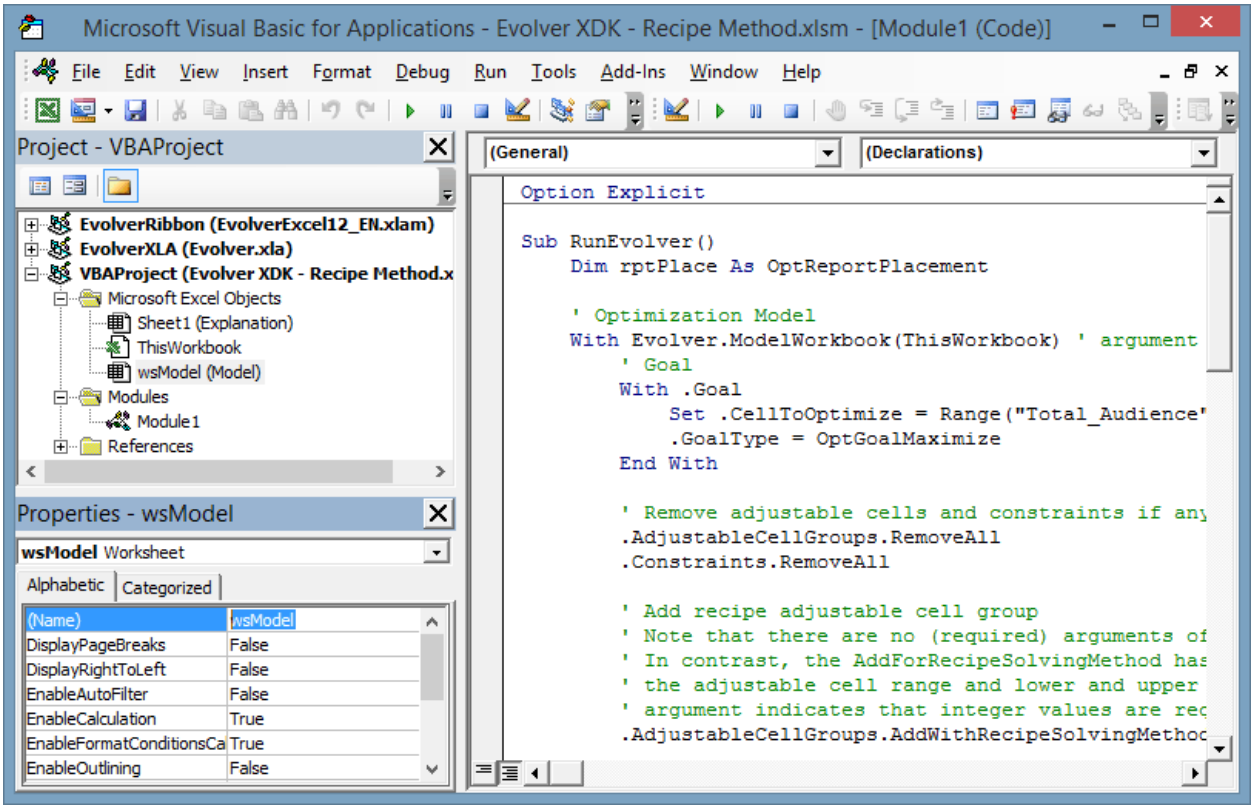
ThisWorkbook

You can always reference a workbook by name, as in Workbooks("My Example File.xlsm"). Alternatively, you can refer to the *active* workbook with the built-in object **ActiveWorkbook**. However, an even safer reference is to **ThisWorkbook**, which always references the workbook containing the VBA code. (It is safer than ActiveWorkbook because the workbook containing the code might not be active when you run your program.)

Worksheet Code Names

The code in the example files sometimes references worksheets by their "code" names. A worksheet actually has two names, the name you see on its tab, and a code name, which can be set only in the Visual Basic Editor. This is illustrated in Figure 6 (see the highlighted line in the Properties section). In this example, the notation wsModel (Model) in the Project section indicates that Model is the name on the tab and wsModel is the code name. (Any names can be used for code names, but a common practice is to use the prefix ws.) One reason for using code names in VBA programs is that you don't need to create a Worksheet object variable; you can simply write wsModel.Range("A1"), for example. A second reason is that if your code refers to the worksheet by name, as in Worksheets("Model"), and someone changes the name on the worksheet tab, an innocent enough change to make, this will break your program. However, because code names can be changed only through the Visual Basic Editor, it is much less likely that anyone will change them.

Figure 6 Code Name for Worksheet



With Blocks

In the XDK example files, you will see a lot of **With** blocks, sections that start with the keyword With and end with the keyword End With. These are used by all good VBA programmers, so you should get used to them. They allow you to avoid a lot of repetitive code, as illustrated by the following block of code. Inside the **With Evolver.ModelWorkbook** block, anything that starts with a period really has **Evolver.ModelWorkbook** to the left of it. For example, **.OptimizationSettings** is equivalent to **Evolver.ModelWorkbook.OptimizationSettings**. As you can see, these With blocks can be nested to avoid even more repetitive code. For example, **.ProgressTrialCount** is equivalent to **Evolver.ModelWorkbook.OptimizationSettings.Runtime.ProgressTrialCount**. It is a *very* good idea to indent these blocks appropriately, as has been done here, for easier reading.

```
    With Evolver.ModelWorkbook
        With .OptimizationSettings
            .Engine.OptimizationEngine = OptEngine_Automatic
            With .Runtime
                .ProgressStoppingCondition = True
                .ProgressMaxChangeIsPercent = True
                .ProgressMaxChange = 0.01
                .ProgressTrialCount = 20000
            End With
        End With
    End With
```

Adding to a Collection

Virtually all collection objects have an Add method for adding an item to the collection. For example, the Worksheets collection has an Add method for adding a worksheet to a workbook. The following code is

typical. It adds a new worksheet after the last worksheet and gives it a couple properties. The Add line is compact in that it not only adds the worksheet (and makes it the *active* worksheet), but it returns a reference to the new worksheet so that the ws object variable can be "set."

```
Dim ws As Worksheet
Set ws = Worksheets.Add(after:=Worksheets(Worksheets.Count))
With ws
    .Name = "Revenues"
    .Range("A1").Value = "Revenues for our new product"
End With
```

If the ws object variable is not needed later in the program, this code can be compacted even further, as follows. Now there is no ws object variable. The With line creates an implicit reference to the new worksheet, so that its properties can be referenced inside the With block.

```
With Worksheets.Add(after:=Worksheets(Worksheets.Count))
    .Name = "Revenues"
    .Range("A1").Value = "Revenues for our new product"
End With
```

A similar construction can be used with Evolver code. The following is typical. The third line uses an "Add" method, AddWithRecipeSolvingMethod, to create a reference to a new adjustable cell group, and then within its With block, adjustable cell ranges are added to the new group.

```
With Evolver.ModelWorkbook(ThisWorkbook)
    ' Add recipe adjustable cell group and two adjustable cell ranges
    With .AdjustableCellGroups.AddWithRecipeSolvingMethod
        If nRoutesConstrained > 0 Then _
            .AdjustableCellRanges.AddForRecipeSolvingMethod _
                flowRangeConstrained, 0, arcCapacityRange
        If nRoutesUnconstrained > 0 Then _
            .AdjustableCellRanges.AddForRecipeSolvingMethod _
                flowRangeUnconstrained, 0, totalCapacity
    End With
End With
```

Intellisense

The reference to the Evolver library mentioned earlier is not only required to make your Evolver code work properly, but it provides you with an invaluable programming aid: Intellisense. As an example, consider the following line:

```
Evolver.ModelWorkbook.Goal.GoalType = OptGoalMaximize
```

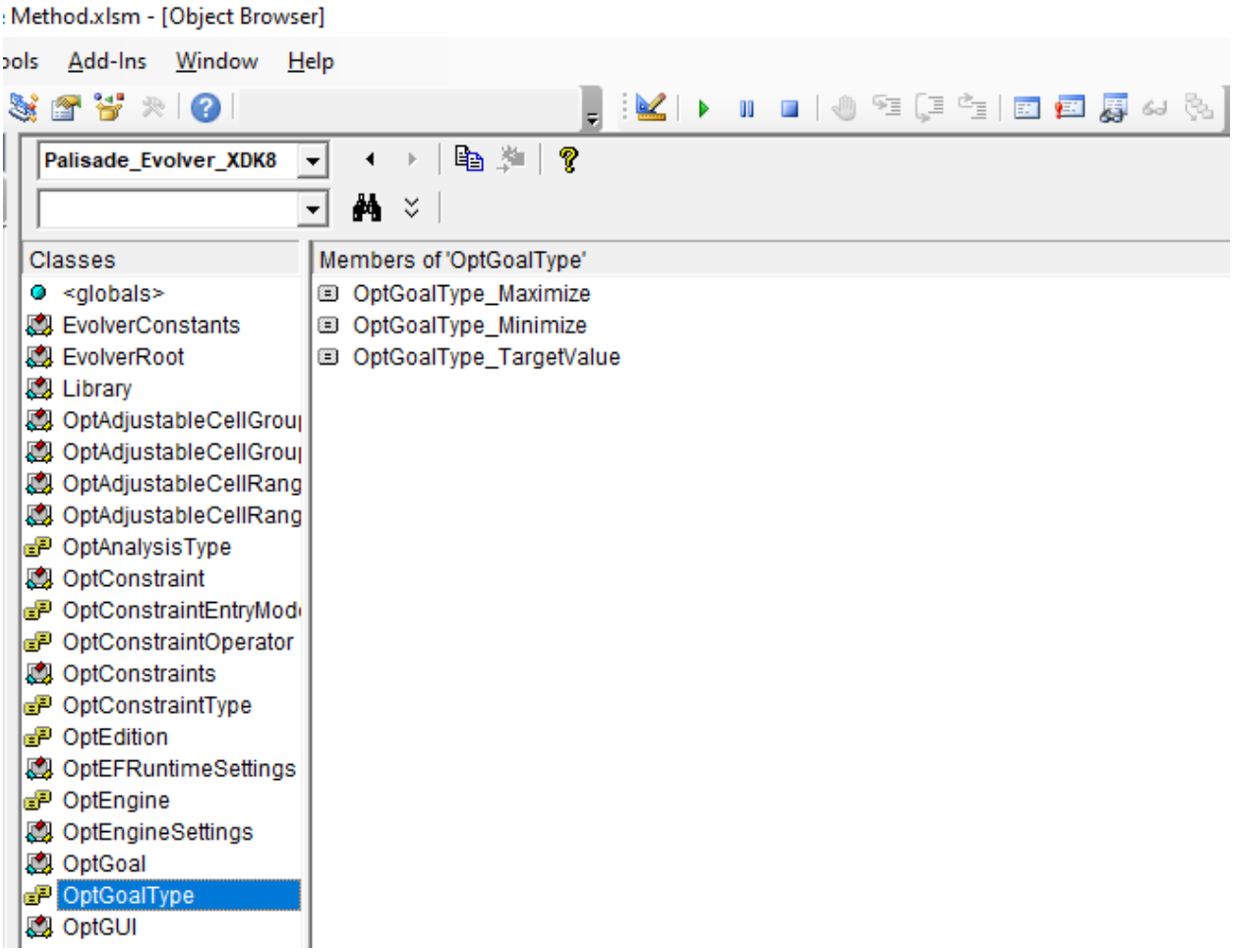
Could you remember all of this? Fortunately, you don't have to. As soon as you type **Evolver.** (including the period), you will see a list you can choose from. Then when you choose **ModelWorkbook** from the list and type a period, you will see another list to choose from. And so on. To cap it off, when you have entered everything to the left of the equals sign and then type the equals sign, you will see a list of the appropriate Evolver constants for the GoalType property that you can choose from. If you are already an Excel programmer, you undoubtedly appreciate the power of Intellisense, and you will be happy to know that it extends to VBA for Evolver. If you are not yet familiar with Intellisense, you are in for a big treat!

Built-In Constants

The line **.GoalType = OptGoalType_Maximize** contains one of many built-in Evolver constants, in this case, OptGoalType_Maximize. All of these constants begin with Opt, and they all have integer values, such as 3 or 445. However, because no person could ever remember these values, they are instead

given meaningful names that programmers can use in their VBA code. You can find the names in the Object Browser (discussed next). For example, Figure 7 lists the constants (on the right) for the three possible settings of the **GoalType** property. Together, this set of three constants is called an **enumeration**. (Strictly speaking, the GoalType property returns the OptGoalType enumeration. This is only one of several enumerations available in Evolver, indicated by the double-yellow icons in the figure. [2])

Figure 7 OptGoalType Enumeration

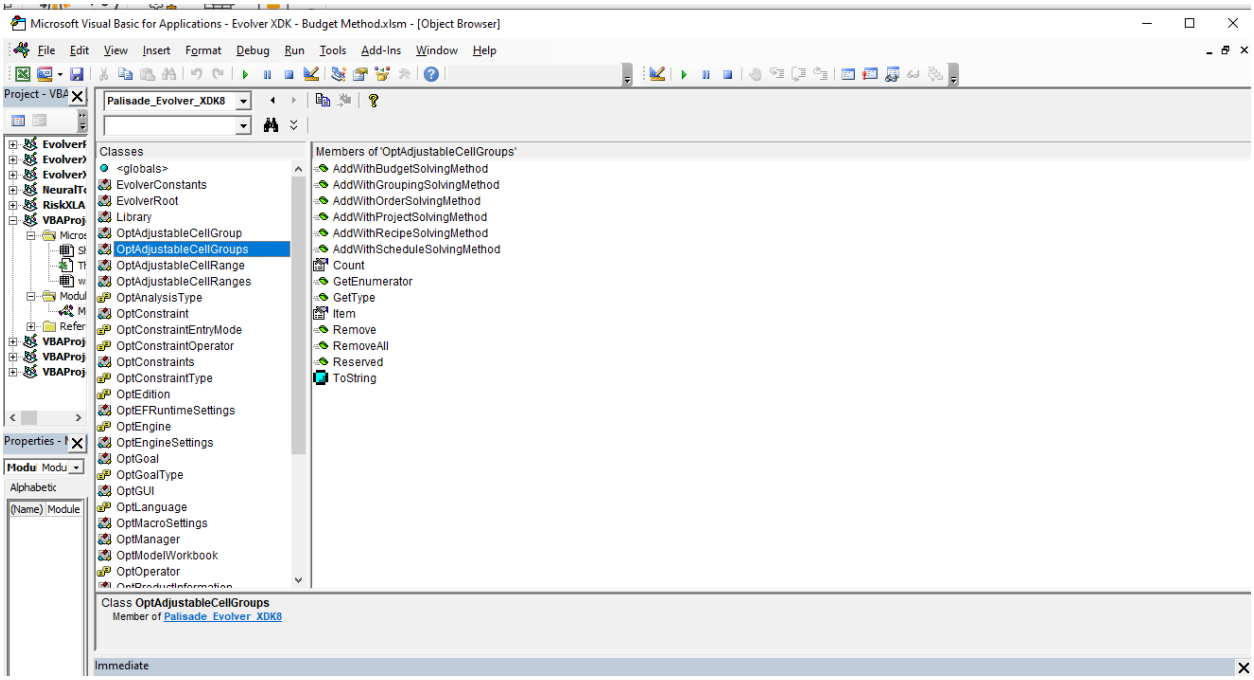


Object Browser

A great source of information is the VBA Object Browser. To see this, make sure a file with a reference to the Evolver library is selected in Project pane. Then click the circled button in Figure 8, and select Evolver from the libraries dropdown list. This Object Browser shows all of the Evolver objects on the left and all of their properties and methods on the right. Also, when you select an item on the right, you get some limited help, such as the list of a function's arguments, at the bottom of the window. Remember that such help is only a few mouse clicks away as you are doing your programming. It often provides just the detail you need to get you past a programming hurdle.

[2] Excel also has many enumerations. Its built-in constants all begin with xl, as in xlDown, xlToRight, and so on.

Figure 8 Visual Basic Object Browser



Appendix

The document to this point contains the information you need to get started with VBA for Evolver. This appendix contains extra information, some of which is rather technical, that might come in handy.

Updating Evolver 5.x, 6.x, 7.x, and 8.x Automation Code to Run in Evolver 8.7

Moving to version 8.7, Palisade has reviewed and improved the XDK interface, taking into account new features and the needs of .NET programmers. This resulted in some changes to the programming interface. The major changes are described here in reference to VBA code, but not every detail is covered. In some cases when code created for earlier versions does not compile in version 8.7, one may need to review the new interface to identify a new way of accomplishing the same objective. Usually it will be possible to identify a similarly named method or property that accomplishes the same task; or in some cases changes to code will be forced by a modified list of method parameters in version 8.7. However, every effort was made to not change the interface unless necessary, and over 95% of the interface is the same as in previous versions.

Updating Library References

If you have Evolver automation code written for a version earlier than 8.7, you'll need to change the references as described before in this guide for that code to work with Evolver 8.7.

OptApplicationSettings Object Eliminated

Version 8.7 no longer contains the Application Settings dialog, and the corresponding **OptApplicationSettings** object was eliminated from the XDK interface. The **ReportPlacement** property that was found in this object is no longer needed. When making the call to generate a worksheet with a report, the programmer has the option of specifying the workbook where the report should be placed.

The remaining report-related properties found in the **OptApplicationSettings** object are no longer supported: **ReportOverwriteExisting** and **ReportReuseNewWorkbook** (report are never overwritten in Evolver 8.7: a new worksheet is always added).

New Format of Enumerated Constant Names

There is a new format for names of enumerated constants in version 8, requiring some changes in VBA code wherever an enumerated constant was used. It should be straightforward to identify the new name of a constant, particularly that the names of the enumerations generally have not changed. For example, **OptGoalType.OptGoalMaximize** in older verisons became **OptGoalType.OptGoalType_Maximize**. In general, full names of enumerations are included as prefixes in the names of the constants, followed by an underscore.

New Methods for Returning Arrays with Optimization Results

In version 8.7 the following two methods that return arrays have new names and somewhat different signatures:

- **GetAdjustableCellValues** became **GetAdjustableCellValuesArray**
- **GetBestValue_EfficientFrontier** became **GetBestValue_EfficientFrontierArray**

Both methods return the **AdjustableCells** array of objects represting the cells whose values are adjusted during an optimiztion, as one of the returned arrays. Before version 8.7 this array was declared as an array of "Object". In version 8.7 it is declared as an array of "Variant". Here is sample code that reads information about adjustable cells after an optimization.

Figure 9 – Example code showing the use of the GetAdjustableCellValuesArray

```
Private Sub PrintAdjustableCells(ByVal modelWorkbook As OptModelWorkbook)
   Dim i As Integer
   Dim adjCells() As Variant
   Dim adjValues() As Double
   Dim rngDestination As Excel.Range

   Set rngDestination = Sheet2.Range("D1") 'Define the range where the values are going to be printed

   'Obtain the adjustable cell values
   modelWorkbook.Results.GetAdjustableCellValuesArray OptWhichSolution_Best, adjCells, adjValues
   'Iterate throuigh each adjustable cell reference and print out its address and its value for the best result found
   For i = 1 To UBound(adjCells)
       rngDestination.Offset(i, 0).Value = adjCells(i).Address
       rngDestination.Offset(i, 1).Value = adjValues(i)
   Next i
End Sub
```

Item no Longer the Default Property

The Item property of collection classes is no longer the default property. So, for example, before version 8.7 adjustable cells specified in the model could be accessed like this, without including the **Item** property explicitly:

```
Evolver.modelWorkbook(ThisWorkbook).AdjustableCellGroups(1).AdjustableCellRanges(1).Ad
justableCells.Address
```

However, in version 8.7 the **Item** property needs to be included explicitly:

```
Evolver.modelWorkbook(ThisWorkbook).AdjustableCellGroups.Item(1).AdjustableCellRanges.
Item(1).AdjustableCells.Address
```

Different Initialization Method with the .NET Interface

See the section of this guide that covers the .NET interface.

Updating Evolver 5.x or 6.x Automation Code to Run in Evolver 8.x

If you have automation code written for Evolver 5.x, 6.x or 7.x, that code should continue to work with Evolver 8.x, *provided that you set the references as described earlier*.

The required references for Evolver 8.x (or 7.x or 6.x) are somewhat different from Evolver 5.x. In 5.x, you made only a single reference to the Evolver 5.x for Excel Developer Kit. If you are converting automation code designed for use with Evolver 5.x to Evolver 8.x, you must remove this old reference and make the two references described earlier.

Also, to automate Evolver version 5.7 in 64-bit Excel, you were required to include a block of code to obtain the root object. This code block is no longer needed and should be removed from your code.

Changes to Evolver XDK in Version 6.x and 7.x

Changes have been made to the Evolver Developer Kit due to the addition of the OptQuest optimization engine.

The class of the root Evolver object was "OptApplication". It has been renamed as "EvolverRoot". Most users will not be affected by this change because there was generally no need to use the name of the class in the user code.

OptAdjustableCellGroup methods/properties moved to OptEngineSettings:

- CrossoverRate > GACrossoverRate
- MutationRate > GAMutationRate
- MutationRateIsAuto > GAMutationRateIsAuto
- MutationRateSelectAuto > GAMutationRateSelectAuto
- OperatorSelect > GAOperatorSelect
- OperatorDeselect > GAOperatorDeselect
- OperatorIsSelected > GAOperatorIsSelected
- OperatorsSelectAll > GAOperatorsSelectAll
- OperatorsSelectDefault > GAOperatorsSelectDefault

OptOptimizationSettings properties moved to OptEngineSettings:

- PopulationSize > GAPopulationSize
- RandomNumberSeed
- RandomNumberSeedIsAuto

OptAdjustableCellGroups methods for adding groups (AddWithRecipeSolvingMethod, AddWithBudgetSolvingMethod, ...):

- ApplyUserDefaults parameter no longer has any effect, since genetic algorithm parameters (mutation rate and crossover rate) are no longer defined separately for each adjustable group

Eliminated properties:

- OptConstraint.DeviationOnCurrentTrial
- OptViewSettings.KeepTrialByTrialLog
- New property (added in relation to the new OptQuest optimization engine):
- OptEngineSettings.OptimizationEngine

Demand-Loading Evolver

You usually want Evolver to load automatically when you open your files containing Evolver macros. By adding the reference to EvolverXLA in the VBA references as described earlier, this will occur. However, there are some instances where this is not the behavior you want. For example, you might want to create an application that uses Evolver only part of the time. Then requiring Evolver to always be loaded is not appropriate. In such cases, the loading and shutdown of Evolver itself is your code's responsibility.

If you remove the EvolverXLA reference, which is really a reference to the Evolver.xlam add-in file, you can still automate Evolver, provided that you load Evolver yourself, but you need to do some extra work to obtain the root Evolver reference.

You can get the root Evolver object in a "late-bound" fashion—that is, without an early-bound reference to EvolverXLA—by using Excel's Application.Run command as follows:

```
Dim Evolver As Palisade_Evolver_XDK8.EvolverRoot
Set Evolver = Application.Run("Evolver.xlam!Evolver")
```

This code assumes that Evolver is already loaded, and it calls the public property **Evolver** in a late-bound fashion to access the root automation object. If you store **Evolver** as a global (non-local) variable, we recommend that you set it to Nothing when you are finished using it for automation. Actually, you could call this variable anything, but by using the variable name **Evolver**, your code will only be minimally changed compared to early-bound code.

Automating Evolver from .NET

Although the Evolver automation library is primarily designed for use from within Excel's VBA language, and all of the code examples of this documentation, excepting this section, are written for VBA, it is also possible to use the .NET environment to control Evolver. Note the custom .NET code must run inside the Excel process, after Evolver has been launched.

To automate Evolver from .NET, you must first add a reference to the Visual Studio project. Specifically, you should browse to the "Program Files (x86)\Palisade\System" folder (or "Program Files\Palisade\System" on 32-bit versions of Windows) and add a reference to the Palisade.Evolver.XDK8.dll assembly. The developers of custom applications need not concern themselves with the availability of this or other Palisade assemblies at runtime; they will be loaded when Evolver is launched in Excel.

It may also be necessary to add a reference to the Excel automation library. Some methods in the XDK return objects from the Excel automation interface, or take such objects as parameters; examples of such objects are a Workbook or a Range. If those methods are used, then a "Microsoft Excel x.0 Object Library" reference has to be added to the project from the COM tab of the Reference Manager dialog.

In the .NET interface all the functionality is available via methods of the Evolver class in the Palisade.Evolver.XDK8.dll assembly; those methods will show as static to C# code and shared to VB .NET code. There is a required initialization call in the .NET interface, to be made before any other calls to the

XDK (unlike in the VBA interface). Assuming the Palisade.Evolver.XDK namespace has been declared with the "using" (C#) or the "Imports" (VB .NET) keywords, the call to make is:

C# Code

```
Evolver.Initialize();
```

VB .NET Code

```
Evolver.Initialize()
```

Programmers who need to convert existing .NET code that was written for Evolver version 8.6 or lower should note some differences in the initialization process. In that version a root "Evolver" object had to be obtained using an Excel automation call (Application.Run); this code should be removed when converting XDK code from version to version 8.7. Also, the Initialize call needs to be added, as it was not required in earlier versions.

The object model for the .NET interface is the same as the one for the VBA interface. Generally, the documentation for the VBA interface should be sufficient when writing .NET code. Here is some sample .NET code which changes the number of trials to run to 2000 and runs an optimization.

C# Code

```
//This code assumes "using Palisade.Evolver.XDK;" line has been added, as well as an appropriate
"using" line to access Excel automation interface.
public void Optimize()
{
   Evolver.get_ModelWorkbook().OptimizationSettings.Runtime.TrialCount = 2000;
   Evolver.Optimize();
}
```

VB .NET Code

```
'This code assumes that "Imports Palisade.Evolver.XDK " line has been added, as well as an
appropriate "Imports" line to access Excel automation interface.
Public Sub Optimize()
   Evolver.ModelWorkbook().OptimizationSettings.Runtime.TrialCount = 2000
   Evolver.Optimize()
End Sub
```