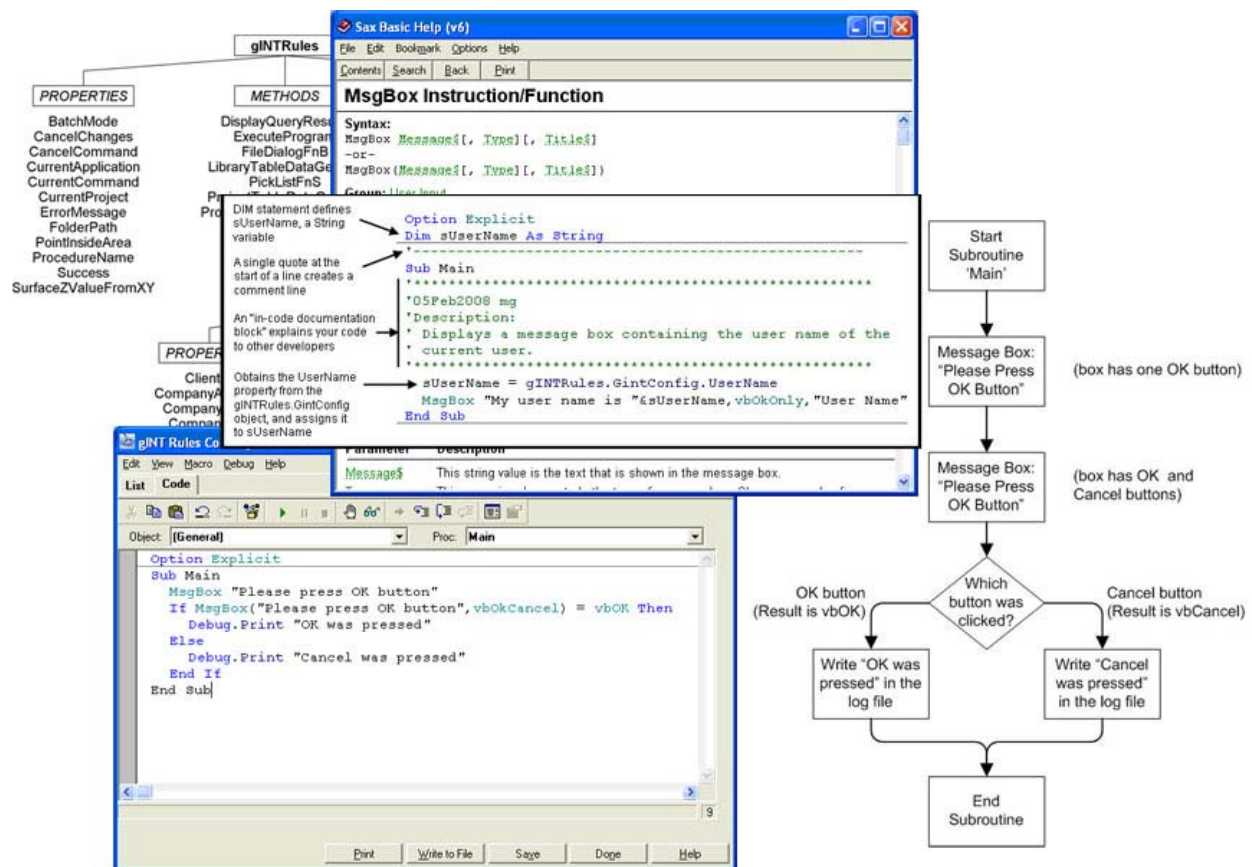


gINT Tutorial

Introduction to gINT Rules



The image displays a screenshot of the gINT software interface, illustrating the implementation of a rule. It includes a code editor, a help window, and a flowchart.

Code Editor (gINT Rules Co):

```

Option Explicit
Dim sUserName As String

Sub Main
    '05Feb2008 mg
    'Description:
    ' Displays a message box containing the user name of the
    ' current user.
    sUserName = gINTRules.GintConfig.UserName
    MsgBox "My user name is " & sUserName, vbOkOnly, "User Name"
End Sub
    
```

Help Window (Sax Basic Help (v6)):

MsgBox Instruction/Function

Syntax:
 MsgBox Message[, Title]
 -or-
 MsgBox (Message[, Type[, Title])

Flowchart:

```

graph TD
    Start([Start Subroutine 'Main']) --> MB1[Message Box: "Please Press OK Button" (box has one OK button)]
    MB1 --> MB2[Message Box: "Please Press OK Button" (box has OK and Cancel buttons)]
    MB2 --> Decision{Which button was clicked?}
    Decision -- "OK button (Result is vbOK)" --> LogOK[Write "OK was pressed" in the log file]
    Decision -- "Cancel button (Result is vbCancel)" --> LogCancel[Write "Cancel was pressed" in the log file]
    LogOK --> End([End Subroutine])
    LogCancel --> End
    
```

The information in this publication is subject to change without notice and does not represent a commitment on the part of gINT Software. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software must be used or copied only in accordance with the terms of the agreement.

Printed November 20, 2008

Disclaimer: Every effort was made to ensure accuracy. However, gINT Software makes no warranty as to the correctness of this information or the supplied files.

All rights reserved worldwide. No part of this publication may be reproduced in any form or by any means without the prior written consent of gINT Software. Comments are welcome and become the property of gINT Software. gINT® is a registered trademark of gINT Software. All other products mentioned are trademarks or registered trademarks of the respective producers.

© Copyright 1985-2007 by gINT Software, Inc.
Printed in the United States of America

7710 Bell Road
Windsor, California 95492-8518
+1 707 838 1271
+1 707 838 1274 (Fax)

Web Site: www.gintsoftware.com
E-mail: support@gintsoftware.com
Sales: sales@gintsoftware.com

Table of Contents

Using this Tutorial.....	1
Setting up Sample Files	1
Overview of gINT Rules	2
Creating a First gINT Rule	3
Invoking a gINT Rule from the Add-Ins Menu	11
Accessing gINT Properties from a Rule	14
A Rule that Changes Group Tab Visibility for Different Users.....	16
How the Group Tab Visibility Rule Works	18
The LibraryOpen Subroutine	18
The CheckApplicationRights Subroutine.....	20
Improving the Group Tab Visibility Rule.....	25
Testing the Code	30

Using this Tutorial

This tutorial is designed for intermediate to advanced level gINT users, and is intended for self-study. It provides an introduction to the gINT Rules programming facility, which enables you to write program code to extend and customize gINT. gINT Rules code can implement data validation, computations, statistics and interpretation, pre-population of data tables, security features, custom menus, and various other customizations. At the conclusion of this tutorial you will have enough familiarity with gINT Rules to try your hand at modifying sample rules available from the gINT website. You may also at that time want to read the more extensive manual, *gINT Rules for Version 8*.

You should be familiar with gINT report design and data design before working through this tutorial, and some experience with programming, while not an absolute requirement, is recommended.

Setting up Sample Files


Before starting the tutorial, you need the following:

- gINT Version 8 installed on your computer
- Sample files installed in the appropriate subfolders of the \gINT\ installation folder (usually C:\Program Files\gINT\)

To obtain and install the sample files, do the following:

1. Go to www.gintsoftware.com/support_doc.html.
2. Click on the link 'Standard Data Files for All gINT Tutorials'.
3. Extract the following files to the indicated locations:

File	Destination
training.glb	\gINT\libraries\
training.gpj	\gINT\projects\

 **Note:** If these files are already present in the indicated locations because of working on another tutorial, you do not need to replace them.

Overview of gINT Rules

gINT Rules is the programming facility in gINT that enables you to write program code to be executed in certain circumstances in **INPUT** and **Library Data**, such as when saving data or in response to selecting a menu option. gINT Rules provides the ability to significantly expand the range of capabilities of the program. The language used is a variant of Visual Basic for Applications® called Sax Basic®, sold by Polar Engineering and Consulting™ (and similar to writing Excel macros). gINT Rules are high-level rules that allow you to go beyond basic field rules and write program code to extend and customize gINT. Some of the main uses of rules are the following:

- **Data Validation:** Providing automatic validation checks of your data, such as ensuring that layers do not overlap or have gaps, or that RQD is less than or equal to sample recovery.
- **Performing Calculations:** Examples include computing lab testing values or generating project statistics.
- **Pre-Populating Data:** On creation of a new table, a rule can be triggered to insert default values, such as sample top depths and types on new sample data for each borehole.
- **Data Interpretation:** You can write rules that generate interpretation of your data, such as generating soil properties based on CPT results, or estimating the date of drilling completion for a project based on the drilling rates to date.
- **Security and Interface Alteration:** Rules can enforce access restrictions to particular tables, fields and reports, or alter how much of the interface is displayed to particular classes of users.
- **Creating Custom Menu Items:** You can create your own menu items in the **Add-Ins** menu, enabling gINT rules you have created to be launched from the menu bar. This enables you to encapsulate multiple-step processes—that would normally require training a person to perform—into a click of a menu option and the answering of simple questions in custom dialog boxes.

While you can learn to use the gINT Rules feature if you are not a programmer, it is programming, and for non-programmers it can appear overwhelming. You may choose to contract with gINT software to perform custom gINT Rules programming rather than take this on yourself. However, the set of exercises in this training module will demonstrate the essentials of creating and modifying gINT rules and implementing them in the interface. It will also provide you with the means to automate some useful tasks with very simple code. If you choose to pursue gINT Rules further, gINT provides extensive documentation and samples that you can customize.



The *gINT Rules for gINT Version 8 Manual*, available from **Help ► Manuals**, is a complete reference on gINT Rules. After completing the present module, the manual is worthwhile reading if you want to obtain a more complete understanding of gINT Rules.



The gINT Software website provides several example applications, including rules code (in the supplied library), sample data (in the project), and instructions for use. Go to http://www.gintsoftware.com/support_gintrules.html, download the desired sample zip file, extract the files to your gINT folders, and merge the library contents into your standard library.



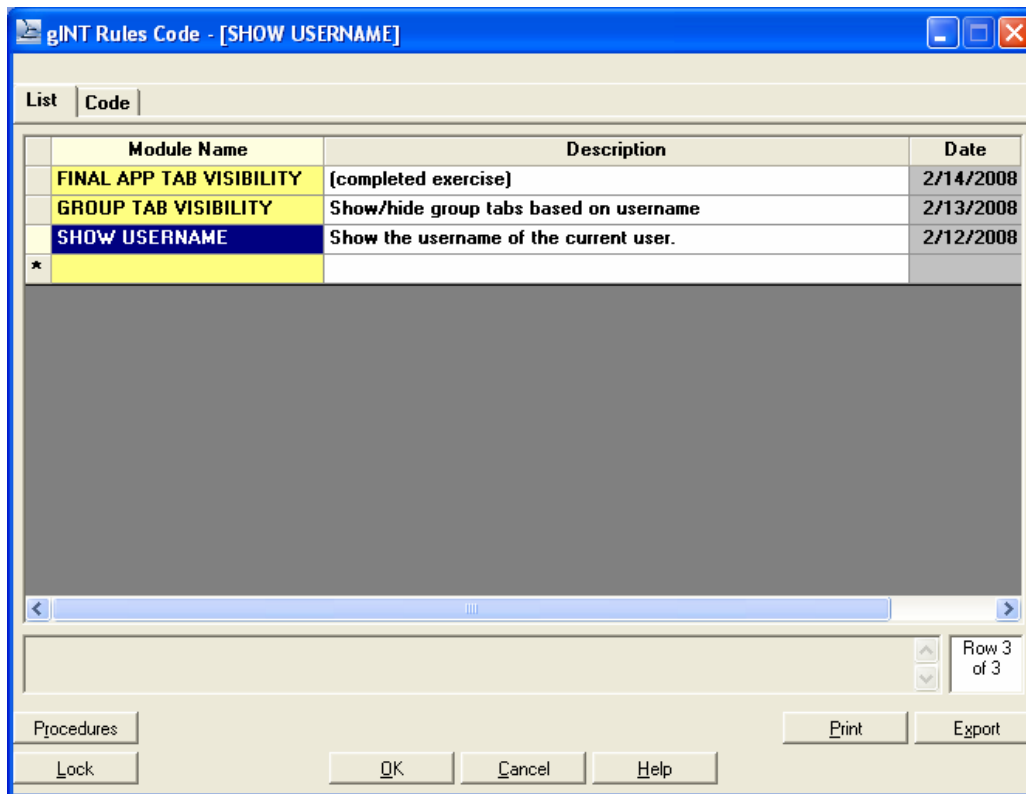
The online help for Sax Basic provides documentation for individual language elements, including syntax, explanations of arguments, and sample code. This is demonstrated in the next section. The Sax Basic help is obtained by opening the gINT Rules window (**gINT Rules ► gINT Rules Code**), clicking the **Code** tab, selecting **Help ► Editor Help ► Search**, then searching for the desired language element.

Creating a First gINT Rule

We will create a simple rule that demonstrates the creation of custom message boxes, and the generation of different program results from different user responses. As will happen often in creating rules, you will copy the initial code from an example, and then make modifications, so that you do not need to do everything from scratch. Perform the following steps:

1. Ensure that the **training.glb** library is active (**File ► Change Library**) and the current project is **training.gpj**.

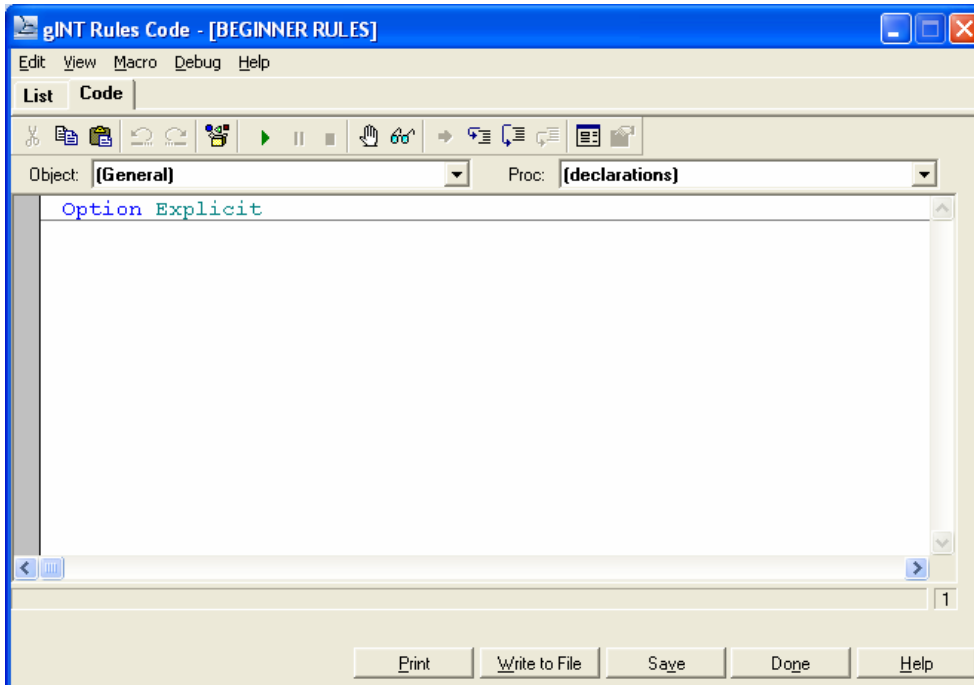
2. Go to **INPUT**. Select **gINT Rules ► gINT Rules Code**. The **List** tab of the **gINT Rules Code** window appears. Three **gINT Rules** modules are already provided; these will be used later.



3. Click inside the left cell in the bottom row (the row preceded by an asterisk).

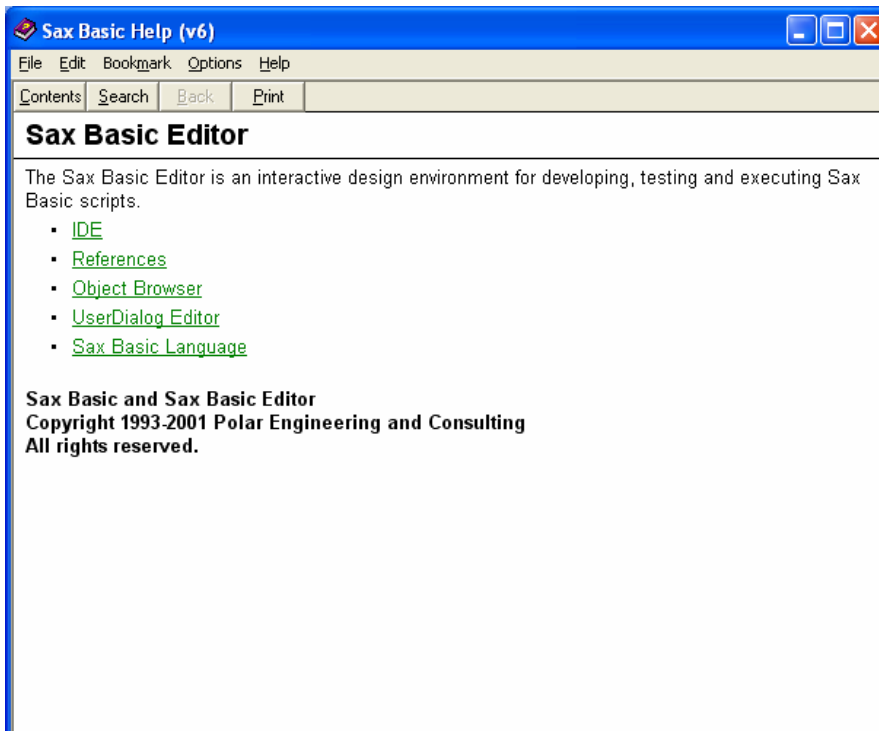
4. For **Module Name**, enter 'BEGINNER RULES', and for **Description**, enter 'Simple rules, including message box manipulation'. This creates a new **gINT Rules** module.

- Click the **Code** tab. The area for creating program code for the new module appears.

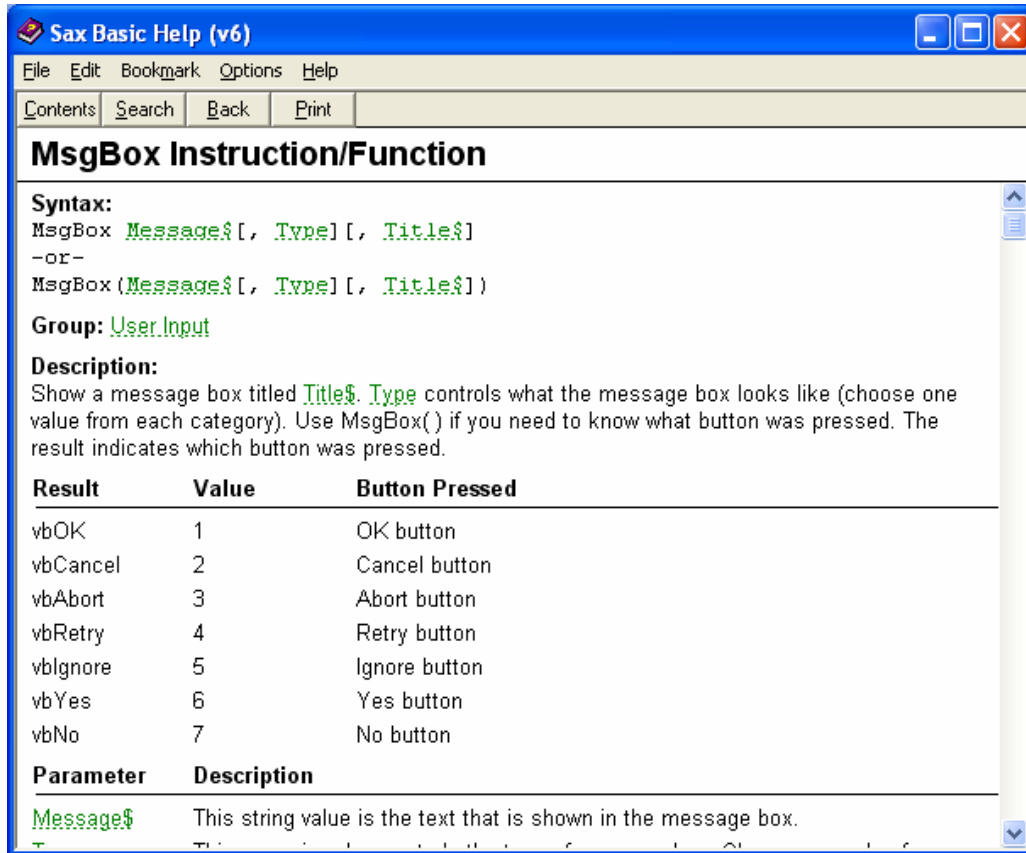


The **List** and **Code** tabs provide alternate views of the rule modules you create. **List** displays all module names and their descriptions. **Code** enables you to create or edit the module whose name is currently highlighted in **List**.

- Select **Help ► Editor Help**. This opens the Sax Basic help window. (Don't click the **Help** button at lower right in the gINT Rules Code window, because that opens gINT Help).



- Click the **Search** tab. In the **Type the First Few Letters of the Word You're Looking For** box, enter 'msgbox', then click **Display**. The documentation for the MsgBox() function is displayed.



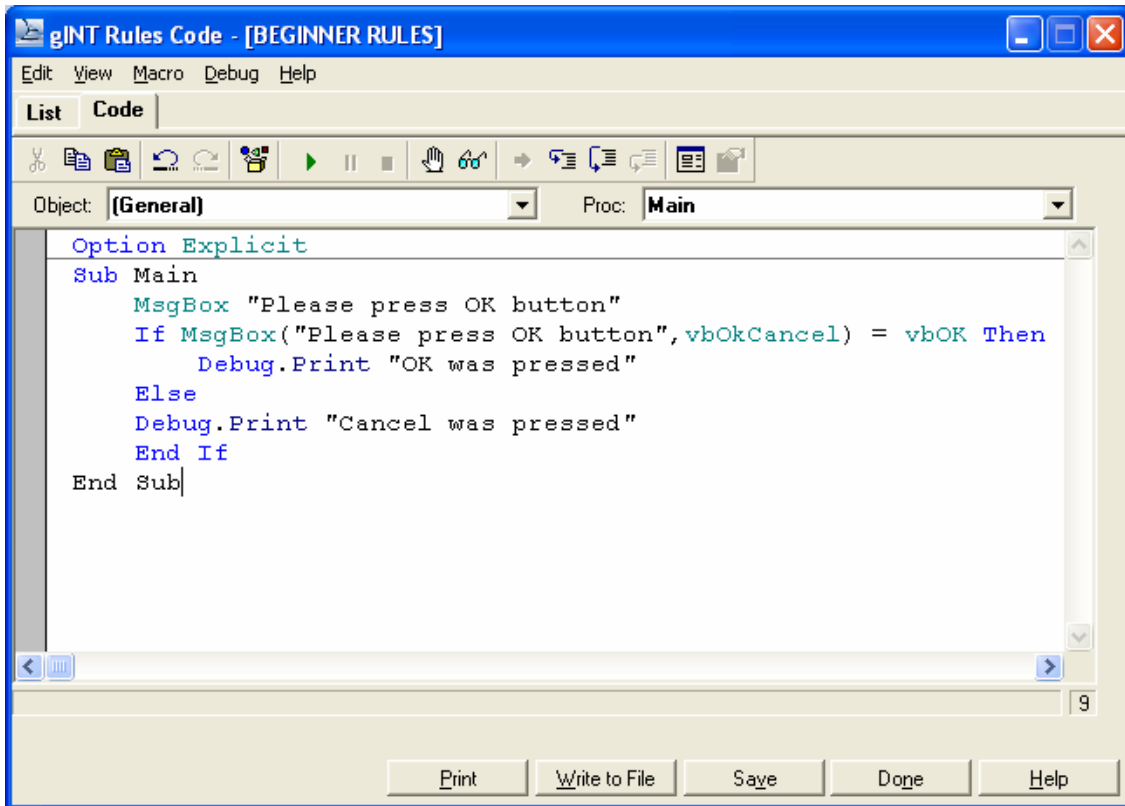
This is a typical online help topic for a function in a programming language. Notice the following elements:

- The **Syntax** section at the top, which specifies the format the function must be in.
- A **Description** section, which explains the purpose and use of the function.
- Explanations of possible **Result** values returned by the function.
- Descriptions of **Parameters** (some required, some optional) that you can or must include in the function statement.


 **Note:** For our purposes, a *parameter* and an *argument* are the same thing.

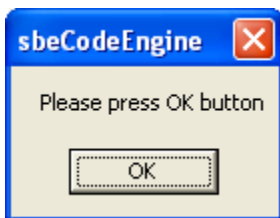
- Scroll down to the bottom of the **Sax Basic Help** window. Notice the **Example** section. Highlight all of the code in this section with the mouse, being careful not to click on any hyperlinks, then press Ctrl-C (copy). We will paste this sample code into our gINT rule.

- Minimize the Sax Basic Help window. In the gINT Rules Code window, click beneath the first line (which reads 'Option Explicit') to start on a new line, then press Ctrl-V (paste). Your rule should appear as shown:

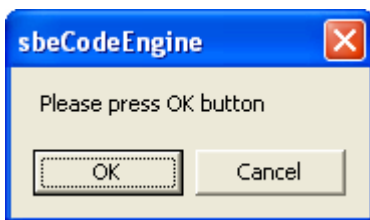


We'll discuss how this code works in a moment. For now, let's see what it does.

- Click the Save button (at the bottom), then click the Play  button in the toolbar. The following message box appears:



- Click OK. Another message box appears:



- Click OK again. This returns you to the gINT Rules Code window.

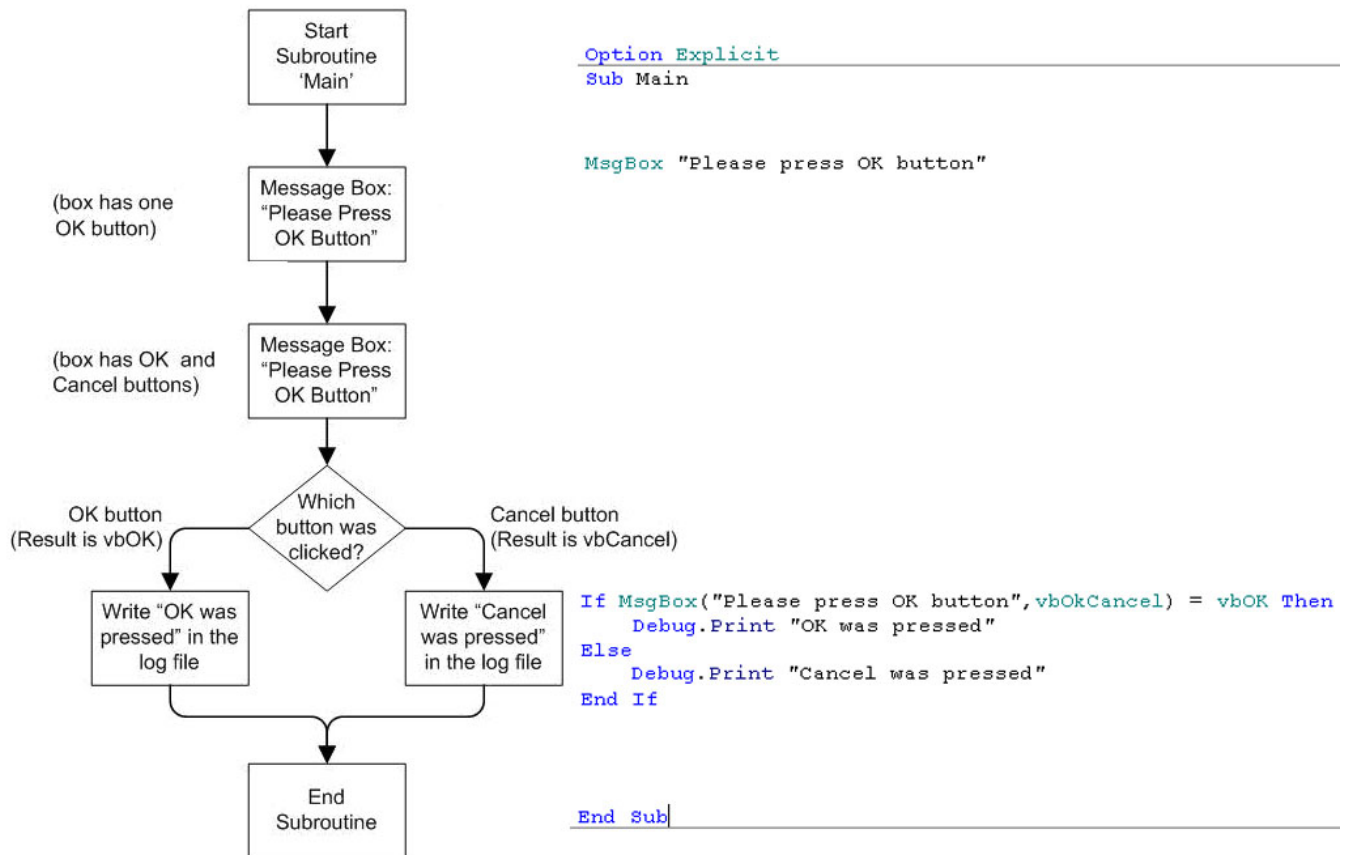
The rule we have created has the following code, which we've annotated:

DECLARATIONS section - creates variables and specifies settings	→	<code>Option Explicit</code>
SUB statement- starts the "subroutine" to perform	→	<code>Sub Main</code>
IF-THEN-ELSE structure containing two MsgBox function calls	→	<code>MsgBox "Please press OK button"</code> <code>If MsgBox("Please press OK button", vbOkCancel) = vbOK Then</code> <code> Debug.Print "OK was pressed"</code> <code>Else</code> <code> Debug.Print "Cancel was pressed"</code> <code>End If</code>
END SUB statement - ends the subroutine	→	<code>End Sub</code>

Note the following points:

- Everything above the first line is the **Declarations** section. You should normally include 'Option Explicit', which is provided by default (it's unimportant at this time as to why). If you create variables (which we will do later), they go in this section.
- Everything between the **Sub** statement and the **End Sub** statement is the *subroutine*. A subroutine is a chunk of code that can be executed as a unit. Our subroutine is called 'Main', because that's the word that appears after **Sub**.

The code within our subroutine proceeds in a certain order, depending on what actions you take. This conditional order of execution of the code is regulated by the IF-ELSE-END IF structure, illustrated as follows (the code appears on the right, what it does is on the left):



There are two different formats available for the **MsgBox** statement: as a command, or as a function. Both are used in our subroutine. The first message box is implemented with the following command:

```
MsgBox "Please press OK button"
```

The command syntax for **MsgBox** is the following (according to the Sax Basic help):

```
MsgBox Message$, [Type][, Title$]
```

For **Message\$**, we have "Please press OK button". We have omitted the **Type** and **Title** parameters, which are optional (indicated by the square brackets around them).

The second message box is implemented as a function (inside an If statement):

```
If MsgBox("Please press OK button", vbOkCancel) = VbOK
```

A function encloses the parameters in parentheses, and returns a result code value. We are launching a message box by executing the `MsgBox()` function, with a `Message$` parameter value of "Please press OK button", and a `Type` value of 'vbOkCancel' (which specifies that the message box should have an **OK** button and a **Cancel** button). The function returns a result code value, which indicates which of the two buttons was clicked—either 'vbOK' if **OK** was clicked or 'vbCancel' if **Cancel** was clicked.

The IF-THEN-ELSE structure that processes the result code was illustrated in the flowchart with a 'Which button was clicked?' decision box (with arms pointing left and right), and the two result boxes beneath it. This code is as follows:

```
If MsgBox("Please press OK button",vbOkCancel) = vbOK Then
    Debug.Print "OK was pressed"
Else
    Debug.Print "Cancel was pressed"
End If
```

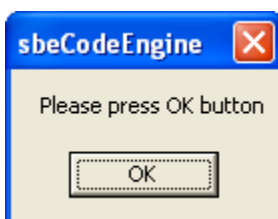
The statement after **Then** is processed if the function returned 'vbOK' because "Then" means "the expression evaluates to TRUE". The statement after **Else** is processed if the function returned 'vbCancel', because "Else" means "the expression evaluates to FALSE", and the expression being tested is "the return code from the `MsgBox()` function is equal to `VbOK`"—which is FALSE, if the return code was 'vbCancel'.

There are some problems with our code that we will correct next. One is that it is improperly formatted, which makes code harder to read and understand. The other is that the **Debug.Print** command used in our two result blocks just creates debugging messages in a status window—we want text to print on the screen.

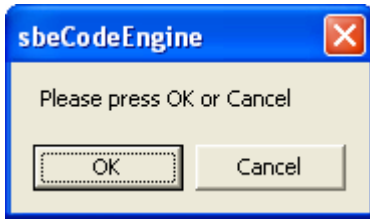
13. In the **gINT Rules Code** window, indent the second **Debug.Print** statement by four characters. Correct use of indenting makes code more understandable, especially when there are IF-THEN-ELSE blocks nested inside each other.
14. Replace the words "Debug.Print" with the word "MsgBox" in both lines where it appears.
15. In the **If** statement, replace the word "button" with the words "or Cancel". The code should now read as follows:

```
If MsgBox("Please press OK or Cancel",vbOkCancel) = vbOK Then
    MsgBox "OK was pressed"
Else
    MsgBox "Cancel was pressed"
End If
```

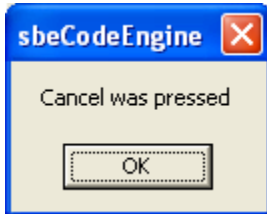
16. Click **Save**, then click the Play  button. You see this message:



17. Click **OK**. You see this:



18. Click Cancel. You see this:



19. Click OK. You are returned to the gINT Rules Code window.

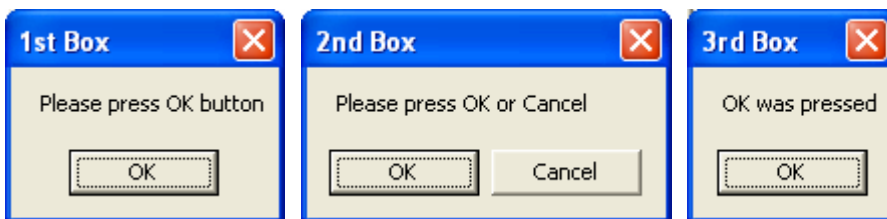
We will make one final change. The default value of "sbeCodeEngine" that appears in the title bar of each message box is not user friendly. To replace this, you use the Title\$ parameter in each MsgBox function or command.

20. Add the indicated title strings, preceded by commas, shown in bold below:

```
Option Explicit
Sub Main
  MsgBox "Please press OK button", "1st Box"
  If MsgBox("Please press OK or Cancel",vbOkCancel, "2nd Box") = vbOK Then
    MsgBox "OK was pressed", "3rd Box"
  Else
    MsgBox "Cancel was pressed", "3rd Box"
  End If
End Sub
```

Don't forget the preceding commas!

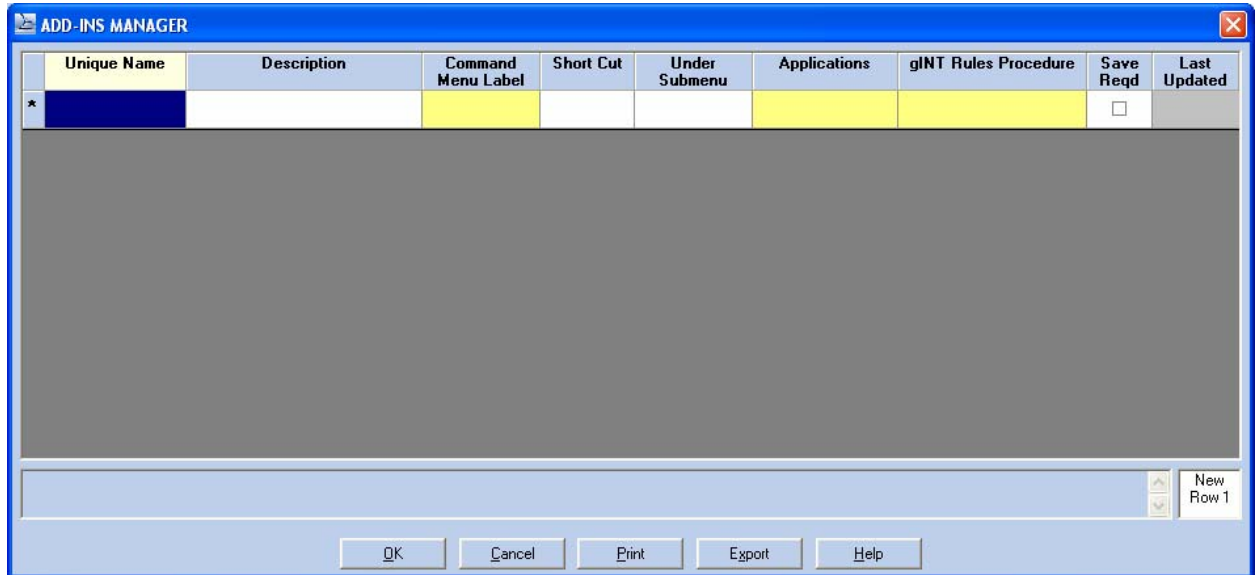
21. Save and play the subroutine again. You see the new title bars.



Invoking a gINT Rule from the Add-Ins Menu

Up until now, we have only executed a gINT rule in the development environment—the gINT Rules Code window. Next, we will set the rule up to be executed from a menu option in gINT, making it accessible from the user interface. A special menu—the **Add-Ins** menu—enables this capability. Perform the following steps:

1. Click **Done** to close the gINT Rules Code window.
2. Select **gINT Rules ► Add-Ins Manager**. The **ADD-INS MANAGER** window appears:



This dialog box is used to create menu entries and submenus in the **Add-Ins** menu in the gINT menu bar.

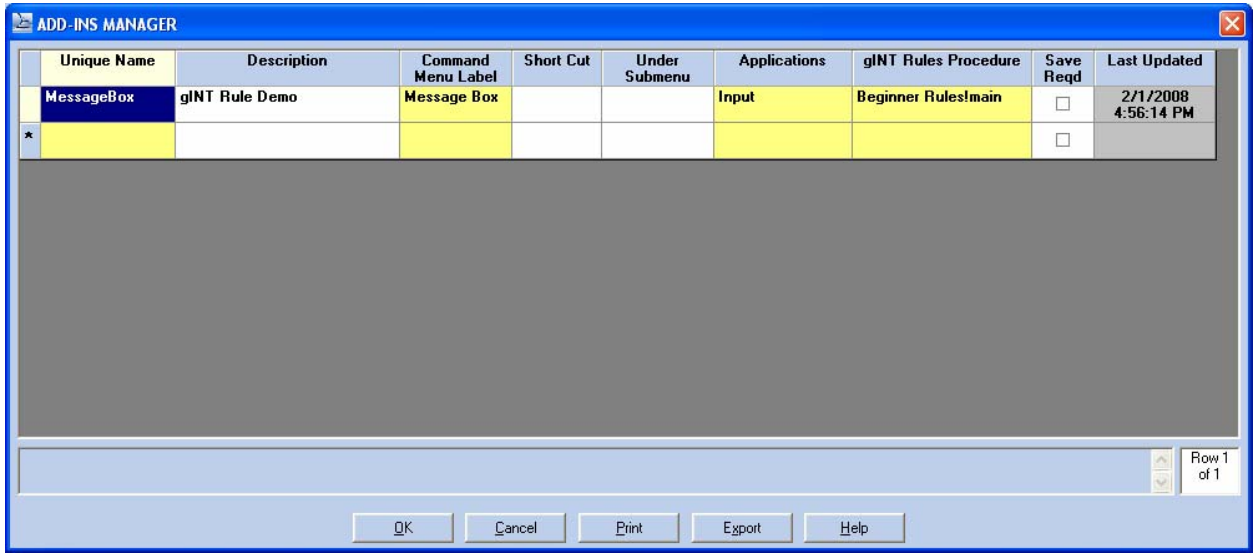
3. Enter the following values:

Field	Value	Comments
Unique Name	MessageBox	
Description	gINT rule demo	
Command Menu Label	Message Box	
Applications	INPUT	
gINT Rules Procedure	Beginner Rules!main	Click the Browse button, select 'BEGINNER RULES' for Type and 'MAIN' for Procedure.

4. Click **OK** to save the **Add-In** menu link and close the window.
5. Click **Add-Ins ► Message Box**. The Message Box rule is executed. Click **OK** on each message box to perform the entire subroutine.

You can also create submenus within the **Add-Ins** menu so that rules of a particular category are assigned to the same submenu. This is especially useful when you have many gINT rules in the **Add-Ins** menu. Do the following:

1. Select gINT Rules ► Add-Ins Manager.



2. In the Under Submenu field for the 'MessageBox' row, enter 'Training'. Click OK.
3. Click the Add-Ins menu. Notice that a Training submenu appears.
4. Hover the mouse pointer over the word 'Training'. Notice that the Message Box option appears.

Next, we'll add an existing gINT rule—one that displays the current user's username—to the Training submenu.

1. Select gINT Rules ► Add-Ins Manager.
2. In the empty bottom row, enter the following values:

Field	Value
Unique Name	ViewUserName
Description	Displays the user name of the current user
Command Menu Label	View User Name
Under Submenu	Training
Applications	INPUT
gINT Rules Procedure	Show Username!main

3. Click OK to save the new Add-In menu item and close the dialog box.
4. Select Add-Ins ► Training ► View User Name.

Notice that there are now two gINT rules under the same submenu (Training). If you create multiple Add-Ins Manager entries with the same Under Submenu value, they all are placed under that submenu. To create other submenus, you enter different Under Submenu values in other rows.

5. Make note of your username (you'll need this in the next exercise), and click OK.

The Add-Ins Manager provides other capabilities, such as the ability to force a data save before a particular menu option is executed, and the ability to specify short-cut keyboard combinations.



For more information, see [Help](#) ► [Index](#) ► [Add-Ins Manager](#).

Accessing gINT Properties from a Rule

The Show Username module illustrates some additional facets of gINT Rules code, including how to access and change property and field values in the gINT environment and database. Do the following:

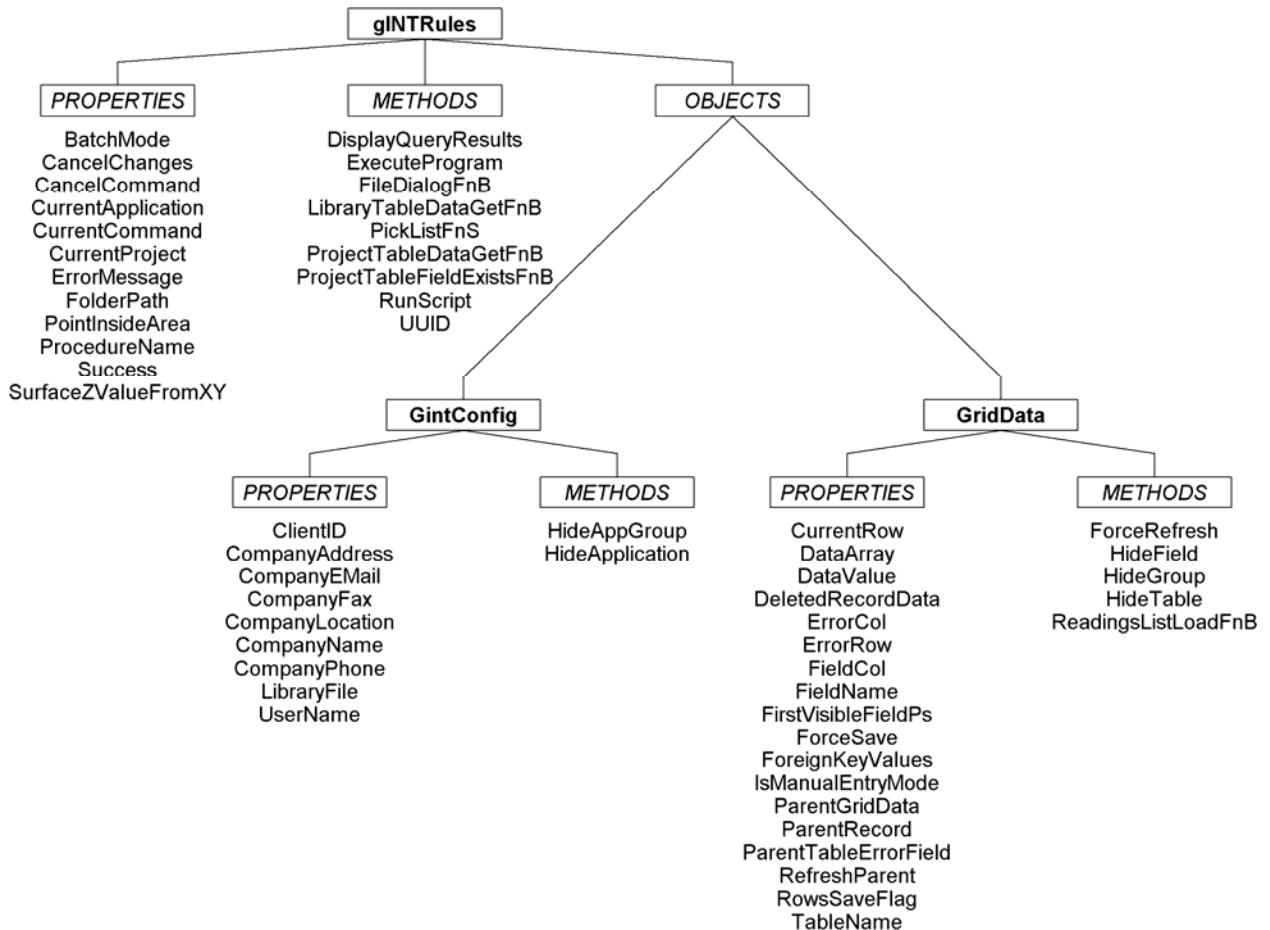
1. Select gINT Rules ► gINT Rules Code. In the List tab, double-click on SHOW USERNAME to open the code tab for that module. The code is as follows:

<p>DIM statement defines sUserName, a String variable</p> <p>A single quote at the start of a line creates a comment line</p> <p>An "in-code documentation block" explains your code to other developers</p> <p>Obtains the UserName property from the gINTRules.GintConfig object, and assigns it to sUserName</p>	<p>→</p> <p>→</p> <p>→</p> <p>→</p>	<pre> Option Explicit Dim sUserName As String ----- Sub Main '***** '05Feb2008 mg 'Description: ' Displays a message box containing the user name of the ' current user. '***** sUserName = gINTRules.GintConfig.UserName MsgBox "My user name is "&sUserName, vbOkOnly, "User Name" End Sub </pre>
---	-------------------------------------	---

2. Notice the following points:

- When you define a new variable for later use, this is done in the DECLARATIONS section using a DIM statement. This is also called *declaring* a variable. You specify the variable's type when you create it—'String' in this case.
- Comment lines start with a single-quote, and do not perform any work except to explain your code to others. Do not underestimate the importance of this to the next person, however.
- The user name is a property of the gINTRules.GintConfig object.

Let's look at the object hierarchy for the gINTRules object and its two child objects, GintConfig and GridData:



The gINTRules object and its two child objects provide *properties* that inform you about the current state of gINT, and *methods* (similar to subroutines) that can be executed to perform actions in gINT.

In the statement

```
sUserName = gINTRules.GintConfig.UserName
```

we are assigning the value of the **UserName** property in the **GintConfig** object to the **sUserName** string variable for later use. However, we cannot reference the **UserName** property or the **GintConfig** object without identifying their places in the object hierarchy. Object hierarchy is expressed by naming the object(s) (and property or method) in top to bottom order, separated by periods.

A Rule that Changes Group Tab Visibility for Different Users

We will utilize properties and methods of the `gINTRules.GintConfig` object to show and hide application group tabs. A gINT Rules module has been included in the library—**GROUP TAB VISIBILITY**—that provides this functionality. We will demonstrate the module in use, study how the code works, then make some modifications that adapt its behavior to new requirements.

Let's set up the **GROUP TAB VISIBILITY** module to perform its role, then test it. Do the following:

1. In the List tab in the **gINT Rules Code** window, double-click 'GROUP TAB VISIBILITY' to display the code for this module.
2. Click the drop-down arrow on the Proc selection list at upper right. Notice the names of the three subroutines in the module ("declarations" is not a subroutine), then select the **LibraryOpen** subroutine from the list. This repositions the editor to the start of the selected subroutine.
3. Read the comments at the top of the **LibraryOpen** subroutine:

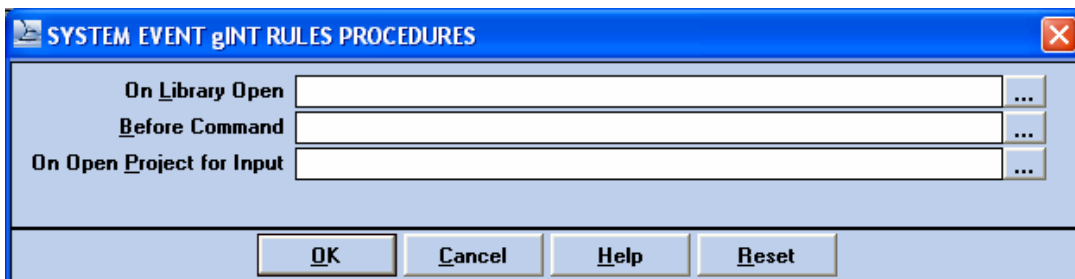
```
'Description:
' Hides application groups based on the user network login name.
' This procedure is called from the gINT Rules:System Events dialog,
' "On Library Open" property.
' Users are given rights in the "GROUP TAB RIGHTS" library table. This library
' table name can be changed in the constant definition in the
' CheckApplicationRights procedure.
' The application groups and/or applications that are defined in that table
' can be changed. The field constants defined below must match those field
' names. If a user is not defined in the library table all the applications
' defined in the table are hidden from the user.
```


From this explanation you can see what **LibraryOpen** is supposed to do, which is the following:

- It is initially invoked by a *system event* called **On Library Open** (we'll have to set this up). This event takes place whenever you either open gINT or change libraries.
- Once launched by the event, it opens the **GROUP TAB RIGHTS** library table and searches for a record with your username in the key field.
- If it finds your record, it notes which group tabs you do not have access rights to (as specified in fields in the library table record).
- It hides each of these group tabs.

We will demonstrate the **LibraryOpen** subroutine next.

4. Close the **gINT Rules Code** window. Select **gINT Rules ► System Events**.



- Click the Browse  button to the right of **On Library Open**. Select a **Type** (module) of 'GROUP TAB VISIBILITY' and a **Procedure** of 'LibraryOpen'. Click **OK** to close the **Module Name** dialog box and again to close the **SYSTEM EVENT gINT RULES PROCEDURES** dialog box.
- Go to **DATA DESIGN ► Library Data**. Select 'GROUP TAB RIGHTS' in the object selector.

	Network User Name	Output	Report Design	Symbol Design	Drawings	Utilities
	someuser	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
*		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

- Enter your username (which you viewed previously by selecting **Add-Ins ► Training ► View Username**) in the **Network User Name** field. Check the **Output** and **Report Design** checkboxes and leave the others unchecked. Click the **Save** icon.
- Close gINT, then reopen gINT. Notice that only the **INPUT**, **OUTPUT**, **DATA DESIGN** and **REPORT DESIGN** tabs appear, and the **SYMBOL DESIGN**, **DRAWINGS** and **UTILITIES** tabs do not appear.
- Go to **DATA DESIGN ► Library Data**. In your record in the **GROUP TAB RIGHTS** library table, put checkmarks in all checkbox fields, then click the **Save** icon.
- Close gINT, then reopen gINT. Notice that all application group tabs are present.

How the Group Tab Visibility Rule Works

Next we will examine the code for the tab visibility rule and see how it works. Perform the following steps:

1. Go to INPUT, then select **gINT Rules ► gINT Rules Code**.
2. Double-click 'GROUP TAB VISIBILITY'. The code for this module appears.
3. Notice that there is a stub subroutine called **Main** in this module, followed by two subroutines with more extensive logic: **LibraryOpen** and **CheckApplicationRights**. Let's look at **Main** first.

```
Public Sub Main
    CallByName Me, gINTRules.ProcedureName, vbMethod
End Sub
```

You should include a **Main** subroutine like this any time you have a multiple-subroutine module that includes subroutines that you want to call from other locations (such as from the **Add-Ins** menu or **System Events**).

What the **CallByName** statement does is obtain the name of the subroutine you want to call by looking it up in the **gINTRules.ProcedureName** system property, and then it invokes that subroutine. [The subroutine name will have been placed in **ProcedureName** by the **Add-Ins** menu item or the system event.] This enables your code to get around a restriction that would otherwise prevent you from calling any subroutine other than **Main** in a module from outside that module.

Note that it is not crucial that you understand why the **CallByName** statement in **Main** works; rather just be aware that if you plan to have more subroutines in a module than just **Main** and to be able to call those subroutines from elsewhere, this **CallByName** statement is what the **Main** routine should contain.



For more information on the **CallByName** statement and the **gINTRules.ProcedureName** property, see the corresponding sections in the *gINT Rules for Version 8* manual.

The LibraryOpen Subroutine

Next we'll look at the subroutine that does the bulk of the work of hiding the application group tabs—**LibraryOpen**.

1. Scroll down to the beginning of the **LibraryOpen** subroutine, which is the 'Public Sub **LibraryOpen**' statement.
2. Since this is a longer module than the ones we have seen so far, drag the top of the code window to the top of your screen, and the bottom of the window to the screen bottom. (You may also need to drag the right edge to the right until nothing is cut off by that edge).

3. Notice the declarations at the top of the subroutine, just after the comments:

```
Const s_Field_Output As String = "Output"
Const s_Field_Reports As String = "Report Design"
Const s_Field_Symbols As String = "Symbol Design"
Const s_Field_Drawings As String = "Drawings"
Const s_Field_Uilities As String = "Utilities"

Dim sUserName As String
```

We're defining five constants (`s_Field_Output`, `s_Field_Reports`, etc.) and one variable (`sUserName`). Constants and variables both store values for later use, but you cannot change the value of a constant. You use a **Const** statement to define a constant and a **Dim** statement to define a variable.

4. Below the declarations, notice the first two program statements that are actually executed.

```
sUserName = gINTRules.GintConfig.UserName
mbFirstCall = True
```

The first of these stores the value of the `gINTRules.GintConfig.UserName` property in the `sUserName` variable. We saw this property in the previous exercise—it has the username with which you are logged on to your computer. We will be passing this argument to a subroutine we'll be calling.

`mbFirstCall` is a TRUE/FALSE variable used as a *flag*. This means that it indicates the state of something until you reset it somewhere else to reflect a change in circumstances. In this case, if the username is not found in the library table where it is going to be looked up, we want to show an error message about this once, not five times (the number of times we're about to call `CheckApplicationRights`). So we set a flag that tells us "is this the first time we've called `CheckApplicationRights`?" Initially we want the answer to be yes (so we set it accordingly) and for subsequent calls we'll change it to no.

5. Notice the remaining code in the subroutine:

```
Call CheckApplicationRights(sUserName, _
                           s_Field_Output, _
                           gr_AppGroup_Output)

Call CheckApplicationRights(sUserName, _
                           s_Field_Reports, _
                           gr_AppGroup_ReportDesign)

Call CheckApplicationRights(sUserName, _
                           s_Field_Symbols, _
                           gr_AppGroup_SymbolDesign)

Call CheckApplicationRights(sUserName, _
                           s_Field_Drawings, _
                           gr_AppGroup_Drawings)

Call CheckApplicationRights(sUserName, _
                           s_Field_Uilities, _
                           gr_AppGroup_Uilities)
```

End Sub

Each of five times we call the `CheckApplicationRights` subroutine, we supply it with the username, the name of the field in the `GROUP TAB RIGHTS` library table to check for that user's record, and the internal name of the application group to disable if access to the corresponding tab is unauthorized for the user in the library table record.

Notice that each statement calling `CheckApplicationRights` is broken into three lines. In order to break one statement onto multiple lines, you must enter the *continuation character*, `'_'`, before each line break.

The CheckApplicationRights Subroutine

Next we'll examine the `CheckApplicationRights` subroutine. Do the following:

1. Scroll down to this routine or select it in the Proc selection list. Read the `Sub` statement and the comments section beneath it:

```
Private Sub CheckApplicationRights(ByVal psUserName As String, _
                                ByVal psRightsField As String, _
                                ByVal piAppGroup As Integer)
'*****
'02Feb2007 sc
'Description:
' Checks the library table listing user rights for rights to
' a specific application or application group.
' If the user is not defined in the table or does not have rights
' to the application or application group, the application or
' application group is hidden.
'Input:
' psUserName - User network login name
' psRightsField - Name of field in library table holding the rights
'                (True/False) for an application group or application.
' piAppGroup - The gINT Rules enumerated value for the application group
'                corresponding to psRightsField.
'Output:
' None
'*****
```

The author of the subroutine has explained in the comments the purpose of the subroutine and of each of the three parameters.

Notice that the `Sub` statement for `CheckApplicationRights` is prefixed by the word `'Private'`, whereas the `Sub` statements for `LibraryOpen` and `Main` were prefixed by the word `'Public'`. `'Public'` specifies that you can call `LibraryOpen` and `Main` from other modules, but `'Private'` specifies that you can't call `CheckApplicationRights` from outside the present module.

Notice also how each parameter is of the form `'ByVal variablename As type'`. The difference between `ByVal` and `ByRef` is a more advanced topic, covered in the *gINT Rules for gINT Version 8* manual—assume for now that you will always use `ByVal`. The `'As type'` clause specifies the type of each parameter passed—two strings and one integer, in this case. Also notice the naming convention used for the parameters—a `'ps'` prefix means “parameter of type string” and a `'pi'` prefix means “parameter of type integer.”

2. View the declarations for the subroutine, just under the comments section.

```
Const s_Personnel_Table As String = "GROUP TAB RIGHTS"

Dim dHaveRights As Double
```

We are declaring one constant and one variable. The string constant `s_Personnel_Table` holds the name of the library table used for the lookup. The `dHaveRights` variable has a data type of Double (double-precision real number) and is used as a parameter in a system function that will be explained shortly.

3. Study the next statement, which is a call to a method (routine) in the `gINTRules` object:

```
If gINTRules.LibraryTableDataGetFnB(s_Personnel_Table, _
                                     psUserName, _
                                     psRightsField, _
                                     "", _
                                     dHaveRights) _
```

The `LibraryTableDataGetFnB` method performs a table lookup in a library table whose name we specify, and returns to us what it finds, if anything. We issue a call to this method, supplying the first three parameters, and obtaining the returned value in either the fourth or fifth parameter, depending on the kind of data we're checking (string or numeric). Here's what the five parameters do:

- **Parameter 1 (supplied, string):** Table name of the library table to access. We've saved this name (GROUP TAB RIGHTS) previously in the `s_Personnel_Table` constant.
- **Parameter 2 (supplied, string):** Record key value to use in the lookup. The key in this library table is the user name, which was passed to the current subroutine (`CheckApplicationRights`) as the parameter `psUserName`. Recall that `psUserName` was originally derived from querying the system variable `gINTRules.GintConfig.UserName`.
- **Parameter 3 (supplied, string):** Name of the field from which to extract data. This is one of the field name constants `s_field_fieldname` that was declared in `LibraryOpen` and passed in the call to `CheckApplicationRights`.
- **Parameter 4 (returned, string):** This parameter returns the contents of the queried field if it contains string data. In our case, we are not obtaining string data, we're obtaining TRUE/FALSE (so-called *Boolean*) data, which is passed in a numeric variable, so we insert the blank string for this parameter as a placeholder.
- **Parameter 5 (returned, double-precision numeric):** This parameter returns the contents of the queried field if it is numeric, which is the case here. Since the queried field will be Boolean, whichever field it is, it will pass a value of zero if FALSE and one if TRUE. Actually, we'll need to convert the returned value from the numeric form in which it is passed to a true Boolean—there is a function that performs this role, as we'll see shortly.

Additionally, the method returns an error code result when executed. This can take the form:

```
errCode = methodname(param1, param2, etc.)
```

where an error code variable is assigned a value of TRUE if the method failed, and FALSE if the method was successful. In the case of our call to `LibraryTableDataGetFnB`, failure is considered to mean the inability to retrieve a library table record, either because the lookup key value (the user name) wasn't located, or a field was referenced that doesn't exist.

Note that the error code variable is not required—the method call can be treated as a TRUE/FALSE expression, as in:

```
If methodname(param1, param2, etc.) THEN
```

In this usage, the method is executed, and then its resulting error code value essentially replaces the method in the course of executing the statement in which it is located. If there was an error, the THEN clause executes (because the “method was true”); if not, the ELSE clause executes (if present).

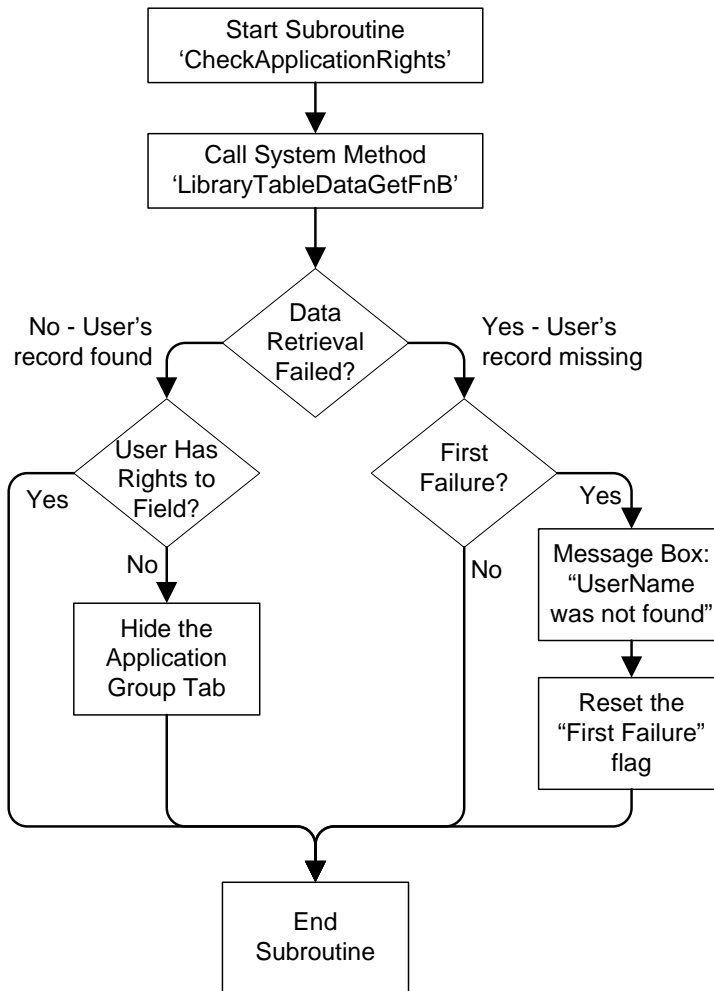
4. Look at the remainder of the code (including the method call we just examined):

```

If gINTRules.LibraryTableDataGetFnB(s_Personnel_Table, _
                                   psUserName, _
                                   psRightsField, _
                                   "", _
                                   dHaveRights) _
Then
  'Failed to obtain data
  If mbFirstCall Then
    'First data retrieval failure
    MsgBox "Your user name (" & psUserName & ") is not defined to the system." _
          & vbCrLf & "Please contact the system administrator.", _
          vbOkOnly, "On Library Open"
    mbFirstCall = False 'Don't show message on subsequent calls.
  End If
Else
  'User's record found
  If Not CBool(dHaveRights) Then
    'Hide the specified application group tab
    gINTRules.GintConfig.HideAppGroup piAppGroup
  End If
End If
End Sub

```

The decision logic in the subroutine is diagrammed in the following flowchart:



Note the following points:

- The `CBool()` function in the expression `CBool(dHaveRights)` is converting a TRUE/FALSE value from a type of Double (double precision real) to Boolean so that it can be used in the IF condition of the IF/THEN statement. It is initially Double because `LibraryTableDataGetFnB` only passes one numeric result and one string one, and the numeric parameter, if used, has to be of type Double.
- The `HideAppGroup` method hides the application group tab specified in its one parameter.

```
gINTRules.GintConfig.HideAppGroup piAppGroup
```

The `piAppGroup` variable has been passed to the current subroutine from the `LibraryOpen` subroutine, where it was initially assigned. It contains a constant with a name that is meaningful to gINT, such as `gr_AppGroup_Output` or `gr_AppGroup_ReportDesign`. It is the `HideAppGroup` method that performs the specific work that we set out for our rule to do, called once for each tab that we need to hide.


Improving the Group Tab Visibility Rule

Now that you've learned how the GROUP TAB VISIBILITY rule works, you can make some modifications to it to change its behavior. A useful improvement would be to be able to turn on and off application tabs within some of the application groups. At present, all users can access **DATA DESIGN**. We did not want to turn this group tab off in case there was a problem with your GROUP TAB RIGHTS library table record—you would be unable to access the library table to fix the problem if you couldn't access **DATA DESIGN**. However, it may not be desirable for all users to have access to all of the **DATA DESIGN** functionality. Let's provide the ability to turn off any of the **DATA DESIGN** application tabs except **Lookup Lists** and **Library Data**.

Do the following:

1. Close the gINT Rules Code window.
2. Go to **DATA DESIGN** ► **Library Tables**, and select 'GROUP TAB RIGHTS' if not already selected.
3. Create five new fields of type Boolean: 'DD: Proj DB', 'DD: Lib Tables', 'DD: Readings', 'DD: User Data', and 'DD: Corresp Files'. Be certain that you type in each of these exactly as shown.

Each of these will be a checkbox. The 'DD' prefix is to indicate that these are tabs within **DATA DESIGN**.

4. Click the Properties  icon and change the **Column Header Lines** to '3', then click OK. This will make the field columns more compact.
5. Click the **Library Data** tab. Verify that the five new fields are present and have checkboxes (if not, return to **Library Tables** and make sure each new field is type Boolean). Also, narrow the columns in **Library Data** so that all fields are visible at once.
6. Ensure that there are checkmarks in all checkboxes except **Drawings**, **DD: Proj Tables**, and **DD: Lib Tables**.

Network User Name	Output	Report Design	Symbol Design	Drawings	Utilities	DD: Proj DB	DD: Lib Tables	DD: Readings	DD: User Data	DD: Corresp Files
mgordon	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
*	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

7. Go to **INPUT**, and select **gINT Rules** ► **gINT Rules Code**. We will copy the 'GROUP TAB VISIBILITY' module to a new module name. Click the **List** tab if it isn't already selected.
8. Click in the **Module Name** cell in the empty bottom row, and enter a **Module Name** value of 'APP TAB VISIBILITY' and a **Description** of 'Show/hide application tabs'.
9. Double-click the 'GROUP TAB VISIBILITY' row to display the code for that module. Select **Edit** ► **Select All** then **Edit** ► **Copy**.
10. Click the **List** tab, then double-click the 'APP TAB VISIBILITY' row. Highlight 'Option Explicit' (we want to replace this statement, not have it appear twice), then select **Edit** ► **Paste** or **Ctrl-V**. The code in the new module is now a duplicate of the code in the old one.
11. Scroll down to the declaration of constants in the **LibraryOpen** subroutine. Enter the following additional **Const** statements (prefixing each with two blank statements, and making sure to exactly enter what is shown here):

```

Const s_Field_DDproj As String = "DD: Proj DB"
Const s_Field_DDlibtab As String = "DD: Lib Tables"
Const s_Field_DDread As String = "DD: Readings"
Const s_Field_DDuser As String = "DD: User Data"
Const s_Field_DDcorr As String = "DD: Corresp Files"

```

Click the **Save** button.

12. Scroll down to the statement calling the **CheckApplicationRights** subroutine for the Utilities tab (this is the last of the five calls to **CheckApplicationRights**, shown below).

```

Call CheckApplicationRights(sUserName, _
                           s_Field_Utilityies, _
                           gr_AppGroup_Utilityies)

```

13. Highlight the three lines of this statement, including the blank line prior to it, then press **Ctrl-C**. Click on the line beneath the statement you copied, and press **Ctrl-V** five times in succession. You should see the following:

```

Call CheckApplicationRights(sUserName, _
                           s_Field_Utilityies, _
                           gr_AppGroup_Utilityies)

Call CheckApplicationRights(sUserName, _
                           s_Field_Utilityies, _
                           gr_AppGroup_Utilityies)

Call CheckApplicationRights(sUserName, _
                           s_Field_Utilityies, _
                           gr_AppGroup_Utilityies)

Call CheckApplicationRights(sUserName, _
                           s_Field_Utilityies, _
                           gr_AppGroup_Utilityies)

Call CheckApplicationRights(sUserName, _
                           s_Field_Utilityies, _
                           gr_AppGroup_Utilityies)

```

14. In the five new **Call CheckApplicationRights** statements, replace 's_Field_Utilityies' with, in order:

- s_Field_DDproj
- s_Field_DDlibtab
- s_Field_DDread
- s_Field_DDuser
- s_Field_DDcorr

[Suggestion: Just replace the 'Utilityies' part, and leave the 's_Field' part alone in each s_Field variable.] The result should appear as follows:

```

Call CheckApplicationRights(sUserName, _
                           s_Field_DDproj, _
                           gr_AppGroup_Uilities)
Call CheckApplicationRights(sUserName, _
                           s_Field_DDlibtab, _
                           gr_AppGroup_Uilities)
Call CheckApplicationRights(sUserName, _
                           s_Field_DDread, _
                           gr_AppGroup_Uilities)
Call CheckApplicationRights(sUserName, _
                           s_Field_DDuser, _
                           gr_AppGroup_Uilities)
Call CheckApplicationRights(sUserName, _
                           s_Field_DDcorr, _
                           gr_AppGroup_Uilities)

```

15. In the five new **Call** statements, replace 'gr_Appgroup_Uilities' with the following, in order:

- gr_App_DataDesign_ProjectDatabase
- gr_App_DataDesign_LibraryTables
- gr_App_DataDesign_ReadingsLists
- gr_App_DataDesign_UserSystemData
- gr_App_DataDesign_CorrespondenceFiles

Your completed changes should appear as follows:

```

Call CheckApplicationRights(sUserName, _
                           s_Field_DDproj, _
                           gr_App_DataDesign_ProjectDatabase)
Call CheckApplicationRights(sUserName, _
                           s_Field_DDlibtab, _
                           gr_App_DataDesign_LibraryTables)
Call CheckApplicationRights(sUserName, _
                           s_Field_DDread, _
                           gr_App_DataDesign_ReadingsLists)
Call CheckApplicationRights(sUserName, _
                           s_Field_DDuser, _
                           gr_App_DataDesign_UserSystemData)
Call CheckApplicationRights(sUserName, _
                           s_Field_DDcorr, _
                           gr_App_DataDesign_CorrespondenceFiles)

```

16. Click **Save**. Scroll down to the bottom of the module. Locate the following line:

```
If Not CBool(dHaveRights) Then
```

17. Create a blank line beneath it (click at the end of the line, then press **Enter**) and enter the following, indented two characters to the right of the IF statement:

```

    'Field is un-checked
    If psRightsField Not Like "DD:*" Then

```

The new comment line is just to clarify why you are in the current block of code. The new **If** statement is creating an additional test that determines whether the checkbox field that is unchecked will turn off an application group tab (field name doesn't start with 'DD:') or an application tab in DATA DESIGN (field name starts with 'DD:'). These require separate processing because different **gINTRules** object methods are used in the two situations.

18. Indent each of the next two lines by two characters each. The result should look like this:

```
If Not CBool(dHaveRights) Then
  'Field is un-checked
  If psRightsField Not Like "DD:*" Then
    'Hide the specified application group tab
    gINTRules.GintConfig.HideAppGroup piAppGroup
```

19. Click at the end of the `gINTRules.GintConfig.HideAppGroup` line and press **Enter** to start a new line. Press **Backspace** so that you are aligned with the `If psRightsField` statement, and enter 'Else'.

20. Press **Enter**, and enter the following (indented two characters to the right of `Else` but on a new line):

```
    'Hide the specified application tab
    gINTRules.GintConfig.HideApplication piAppGroup
```

21. Press **Backspace** so that you are aligned with the `Else` statement, and enter 'End If'. The completed `If Not CBool(dHaveRights)` block should appear as follows:

```
If Not CBool(dHaveRights) Then
  'Field is un-checked
  If Not psRightsField Like "DD:*" Then
    'Hide the specified application group tab
    gINTRules.GintConfig.HideAppGroup piAppGroup
  Else
    'Hide the specified application tab
    gINTRules.GintConfig.HideApplication piAppGroup
  End If
End If
```

The completed **CheckApplicationRights** subroutine should appear as follows:

```
Private Sub CheckApplicationRights(ByVal psUserName As String, _
                                ByVal psRightsField As String, _
                                ByVal piAppGroup As Integer)
'*****
'02Feb2007 sc
'Description:
' Checks the library table listing user rights for rights to
' a specific application or application group.
' If the user is not defined in the table or does not have rights
' to the application or application group, the application or
' application group is hidden.
'Input:
' psUserName - User network login name
' psRightsField - Name of field in library table holding the rights
'                (True/False) for an application group or application.
' piAppGroup - The gINT Rules enumerated value for the application group
'                corresponding to psRightsField.
'Output:
' None
'*****
Const s_Personnel_Table As String = "GROUP TAB RIGHTS"

Dim dHaveRights As Double
'-----

If gINTRules.LibraryTableDataGetFnB(s_Personnel_Table, _
                                    psUserName, _
                                    psRightsField, _
                                    "", _
                                    dHaveRights) _

Then
'Failed to obtain data
If mbFirstCall Then
'First data retrieval failure
MsgBox "Your user name (" & psUserName & ") is not defined to the system." _
        & vbCrLf & "Please contact the system administrator.", _
        vbOkOnly, "On Library Open"
mbFirstCall = False 'Don't show message on subsequent calls.
End If
Else
'User's record found
If Not CBool(dHaveRights) Then
'Field is un-checked
If Not psRightsField Like "DD:*" Then
'Hide the specified application group tab
gINTRules.GintConfig.HideAppGroup piAppGroup
Else
'Hide the specified application tab
gINTRules.GintConfig.HideApplication piAppGroup
End If
End If
End If

End Sub
```

22. Check your work carefully against the final version shown here, then click Save.

Testing the Code

Any time you change program code you should test that code in a way that 1) is harmless to the system and data, and 2) reveals whether things are working correctly (or if not, what is wrong).

Rather than launch the `LibraryOpen` routine immediately from `System Events`, we'll create an `Add-Ins` menu item for it for testing. Also, we'll insert some diagnostic messages in the code so that we can determine what the code is doing as it runs. Do the following:

1. Scroll to the bottom of the module.
2. After the `If Not psRightsField Like "DD:*" Then` line create a new line and enter the following:

```
MsgBox "Hide Group Tab: " + psRightsField
```

This diagnostic message will appear every time a deselected group tab value is encountered, and identifies the field.

3. Copy this new line and insert a copy of it beneath the next `Else` statement. Change 'Hide Group Tab' to 'Hide Application Tab'. When complete, the IF-THEN-ELSE block should look like this:

```
If Not psRightsField Like "DD:*" Then
  MsgBox "Hide Group Tab: " + psRightsField
  'Hide the specified application group tab
  gINTRules.GintConfig.HideAppGroup piAppGroup
Else
  MsgBox "Hide Application Tab: " + psRightsField
  'Hide the specified application tab
  gINTRules.GintConfig.HideApplication piAppGroup
End If
```

4. Close the `gINT Rules Code` window. Select `gINT Rules ► Add-Ins Manager`.
5. Create a new Add-In menu item with a **Unique Name** and **Command Menu Label** of 'Test App Tab', an **Application** of 'INPUT' and a **gINT Rules Procedure** of 'App Tab Visibility!Libraryopen'. Click **OK**.
6. Select `Add-Ins ► Test App Tab`. You should see (and dismiss) a series of message boxes with the following messages:

- Hide Group Tab: Drawings
- Hide Application Tab: DD: Proj DB
- Hide Application Tab: DD: Lib Tables

If you see something else than these three messages, you have some work to do! However, this is standard practice in programming: set up diagnostic messages in useful places, test the code, and search for and resolve problems until the code performs correctly. Then you can go "live" with the code.

7. Go to `gINT Rules ► System Events`. For `On Library Open`, specify 'App Tab Visibility!LibraryOpen' if your code works correctly, or 'Final App Tab Visibility!LibraryOpen' otherwise. Click **OK**.
8. Close and reopen `gINT`. Click **OK** to dismiss each of the three message boxes.

Notice that the **DRAWINGS** tab is hidden.

9. Click on the **DATA DESIGN** tab. Notice that the **Project Database** and **Library Tables** tabs are hidden. This is the desired behavior.
10. Go to **INPUT**, then select **gINT Rules ► System Events**. Clear the value in **On Library Open**, and click **OK**.

☞ **Note:** In practice, once you've got your gINT rule working correctly in production, you would remove the diagnostic messages from the code and the test menu item from the **Add-Ins** menu.