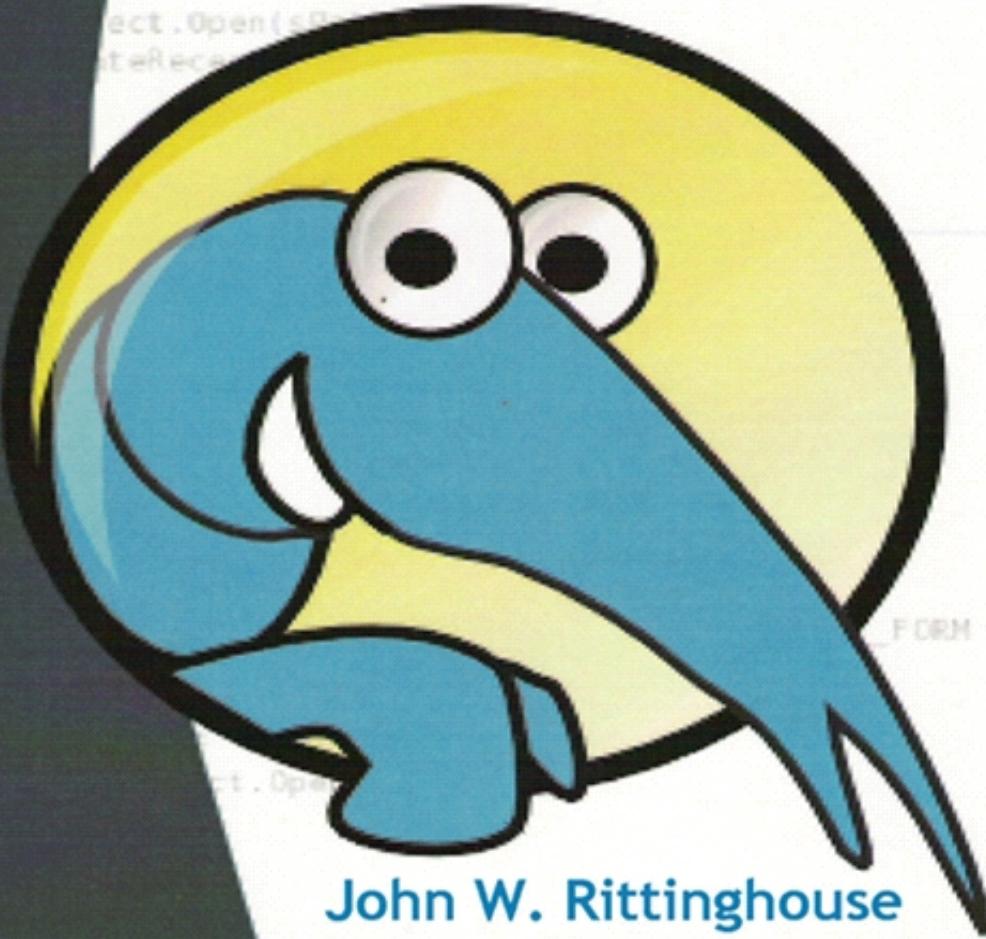


PROGRAMMING

A Beginner's Guide to **GAMBAS**

Revised for Version 3



John W. Rittinghouse
Jon Nicholson

A Beginner's Guide to Gambas – Revised Edition

Copyright Notice for the printed version of this work:

A Beginner's Guide to Gambas Version 3 (this work) is copyright © 2005-2011 by John W. Rittinghouse, all rights are reserved. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the author, John W. Rittinghouse. The author grants a perpetual license to the Gambas user-community for use of the electronic version of this printed work under the terms and conditions of the OpenContent License printed on the following page.

A Beginner's Guide to Gambas – Revised Edition

Copyright Notice for the electronic (online) version of this work, based on the OpenContent License (OPL), Version 1.0, July 14, 1998.

This document outlines the principles underlying the OpenContent (OC) movement and may be redistributed provided it remains unaltered. For legal purposes, this document is the license under which OpenContent is made available for use. The original version of this document may be found at <http://opencontent.org/opl.shtml>.

LICENSE

Terms and Conditions for Copying, Distributing, and Modifying

Items other than copying, distributing, and modifying the Content with which this license was distributed (such as using, etc.) are outside the scope of this license.

1. You may copy and distribute exact replicas of the OpenContent (OC) as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the OC a copy of this License along with the OC. You may at your option charge a fee for the media and/or handling involved in creating a unique copy of the OC for use offline, you may at your option offer instructional support for the OC in exchange for a fee, or you may at your option offer warranty in exchange for a fee. You may not charge a fee for the OC itself. You may not charge a fee for the sole service of providing access to and/or use of the OC via a network (e.g. the Internet), whether it be via the world wide web, FTP, or any other method.

2. You may modify your copy or copies of the OpenContent or any portion of it, thus forming works based on the Content, and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified content to carry prominent notices stating that you changed it, the exact nature and content of the changes, and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the OC or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License, unless otherwise permitted under applicable Fair Use law.

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the OC, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the OC, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it. Exceptions are made to this requirement to release modified works free of charge under this license only in compliance with Fair Use law where applicable.

3. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to copy, distribute or modify the OC. These actions are prohibited by law if you do not accept this License. Therefore, by distributing or translating the OC, or by deriving works herefrom, you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or translating the OC.

NO WARRANTY

4. BECAUSE THE OPENCONTENT (OC) IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE OC, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE OC "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK OF USE OF THE OC IS WITH YOU. SHOULD THE OC PROVE FAULTY, INACCURATE, OR OTHERWISE UNACCEPTABLE YOU ASSUME THE COST OF ALL NECESSARY REPAIR OR CORRECTION.

5. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MIRROR AND/OR REDISTRIBUTE THE OC AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE OC, EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Acknowledgements

I would like to thank Dr. John Rittinghouse for the amazing opportunity to contribute to this book, A Beginner's Guide to GAMBAS, Revised Edition, from the "learner's" point of view. I began by buying his book and offering some ideas for a future revised edition, and turned into a contributor to the revised edition -- evidence of an astounding absence of vanity on his part. I especially want to dedicate my portion of the work on the Second Edition to my son, Ian, whose inquiring mind and ceaseless experiments with operating systems introduced me to Linux and to the Open Source world.

As a long-time Basic programmer, I was elated to find the Gambas version of Basic available for Linux and FreeBSD, with ongoing development for other operating systems. Benoît Minisini and his colleagues, who have brought us the Gambas language free of charge, with its outstanding IDE and component extensions, are to be congratulated and thanked for their continuing work. A small money contribution to the project would not be a bad idea if you can do so.

Finally, I want to express my appreciation for all of the programmers and end-users who write code for open source projects and applications, use open source software, file bug reports, provide help to others on online forums, and contribute money to projects where it's needed. Obviously, as exemplified by the Gambas programming language and its associated community, Open Source engages some of the best talent and best effort available.

Jon Nicholson
March, 2011

It has been nearly five years since I first published Gambas for Beginners. As one of the more aspiring projects in the open source community, I have had the privilege of monitoring its progress and seeing it develop from its infancy to become a reliable programming resource in the Linux community. While I had considered doing an update for some time, for various reasons, I was not able to do so. It was a pleasant surprise when Jon Nicholson contacted me to ask about the quirks he had found in the book as the product transcended from version 1.0 to its current state. After some discussion, I invited Jon to provide comments and feedback as he went through the book. In a very short time, it became clear that Jon was able to do so much more. I invited him to co-author the revision and rolled up my sleeves with him to go back to work on this project. He has been a great resource and strong contributor and has provided innumerable suggestions and a new, fresh perspective to the understanding of the language from the eyes of an absolute beginner. Chapter by chapter, line by line, Jon has reviewed content and code to ensure we have provided the most accurate, updated information available for programming Gambas.

A Beginner's Guide to Gambas – Revised Edition

As with the previous version of this book, I made a commitment to provide an open source version to the Gambas programming community. Details of where this electronic version can be obtained for free are provided at the back of the book. My intent in supporting the Gambas community has been to help those new to the language to muddle through the myriad, poorly documented web sites and approach learning the language in a structured and meaningful manner. The example programs in this book are designed to quickly build programming skills and maximize the understanding of the Gambas language and all of the power it provides to the open source community. I hope you find this reference worthy of your collection.

John W. Rittinghouse, Ph.D., CISM, CRISC
March, 2011

Table of Contents

Acknowledgements.....	3
Chapter 1 - Introducing Gambas.....	13
Gambas Architecture.....	13
The Gambas Programming Environment.....	15
Gambas IDE Components.....	18
Chapter 2 – Language Concepts.....	24
Gambas Variables, Data-types and Constants.....	25
Variable Assignment.....	29
Assignment Using The WITH Statement.....	30
Operators and Expressions.....	30
Arithmetic Operators.....	31
Comparison operators	31
Let's Start Coding Gambas.....	31
String Operators.....	43
Chapter 3 - Keywords and Program Flow Control.....	45
The PRINT Statement.....	45
The IF Statement.....	46
The SELECT / CASE Statement.....	48
GOTO and LABELS.....	49
The FOR / NEXT Statement.....	49
DO [WHILE] LOOP.....	50
WHILE [Expression] WEND Loops	52
The REPEAT UNTIL loop	52
Defining and Using Arrays in Gambas.....	53
Collections.....	55
The FOR EACH Statement.....	55
Chapter 4 – The Gambas ToolBox.....	57
The Button Control.....	59
Common Control Properties.....	60
Button Methods.....	68
Button Events.....	76
The Picture Class.....	76
Chapter 5 – Controls for User Input.....	78
TextLabel.....	80
TextBox.....	82
ComboBox.....	84
ListBox.....	87
Frame.....	89
ToggleButton.....	90
Checkbox.....	91
Panel.....	92

A Beginner's Guide to Gambas – Revised Edition

RadioButton.....	93
Chapter 6 – Menus, Modules, Dialogs and Message Boxes.....	95
The Gambas Menu Editor.....	96
Building Menus.....	99
Dialogs.....	101
Modules.....	103
MessageBoxes.....	111
Information Messages.....	111
Query/Confirm Messages.....	113
Error Messages.....	114
Warning or Alert Messages.....	114
Delete Messages.....	115
Dialog Class File-related Functions.....	116
Dialog OpenFile Function.....	116
Dialog SaveFile Function.....	117
Dialog SelectDirectory Function.....	118
Complete Example Listing.....	119
MMain.module listing.....	122
Chapter 7 – Strings and Data Conversion.....	124
String Functions.....	124
Len.....	124
Upper\$/Ucase\$/Ucase - Lower\$/Lcase\$/Lcase	125
Trim\$, LTrim\$, and RTrim\$.....	126
Left\$, Mid\$, and Right\$	127
Space\$.....	128
Replace\$	129
String\$	130
Subst\$	130
InStr.....	132
RInStr.....	133
Split.....	134
Data Conversion.....	135
Asc and Chr\$	135
Bin\$.....	136
CBool	137
CByte.....	137
CDate.....	138
CFloat.....	139
CInt / Cinteger and CShort	139
CStr / CString	140
Hex\$.....	141
Conv\$	141
Val and Str\$	142
Str\$	142
Format\$.....	145

A Beginner's Guide to Gambas – Revised Edition

Datatype management	147
TypeOf.....	148
Chapter 8 – Advanced Controls.....	149
IconView Control.....	149
ListView Control.....	158
Using the GB Icon Edit Tool.....	162
The TreeView Control.....	164
The GridView Control.....	171
The ColumnView Control.....	175
Layout Controls – HBox, VBox, HPanel and Vpanel.....	179
HBox and VBox.....	180
HPanel and Vpanel.....	180
The TabStrip Control.....	184
Chapter 9 – Working with Files.....	191
Access.....	191
Dir.....	192
Eof.....	193
Exist.....	193
IsDir.....	194
Stat	194
Temp / Temp\$	195
OPEN and CLOSE.....	196
LINE INPUT.....	197
READ, SEEK, WRITE and FLUSH.....	198
COPY, KILL and RENAME.....	200
MKDIR, RMDIR.....	200
Chapter 10 – Math Operations.....	220
Precedence of Operations.....	220
Abs.....	220
Acs / ACos.....	221
Acsh / ACosh.....	221
Asn / ASin.....	222
Asnh / ASinh.....	222
Atn / ATan.....	223
Atnh / ATanh.....	224
Cos.....	224
Cosh.....	225
Deg and Rad.....	225
Exp.....	226
Fix and Frac.....	227
Int.....	227
Log.....	228
Log10.....	229
Max and Min.....	229
Pi.....	230

A Beginner's Guide to Gambas – Revised Edition

Randomize and Rnd.....	230
Round.....	232
Sgn.....	232
Sin.....	233
Sinh.....	233
Sqr.....	234
Tan.....	234
Tanh.....	235
Chapter 11 – Object-Oriented Concepts.....	242
Fundamentals of Object Oriented Programming.....	243
Objects.....	243
Data Abstraction.....	244
Encapsulation.....	244
Polymorphism.....	244
Inheritance.....	244
The Gambas Approach to OOP.....	245
Gambas Classes.....	245
Sample program: Contacts.....	246
The Contact class.....	246
Contact.GetData Method.....	247
Contact.PutData Method.....	249
FMain.class file.....	252
FMain Constructor.....	253
Form_Open Subroutine.....	254
Adding Controls to FMain.Form.....	255
UpdateForm() Subroutine.....	257
Validating User Input.....	263
Adding a Search Feature.....	265
Updating an Existing Record.....	267
Deleting a Contact Record.....	268
Saving Data.....	270
Chapter 12 – Learning to Draw.....	272
Draw Properties.....	272
Background and Foreground.....	272
Clip.....	273
FillColor, FillStyle, FillX, FillY.....	273
Font.....	274
Invert.....	274
LineStyle/LineWidth.....	274
Transparent.....	275
Draw Methods.....	275
Text/TextHeight/TextWidth.....	277
Draw Primitives: Point/Rect/Ellipse/Line.....	278
Draw Primitives: Polygon and Polyline.....	285
Image/Picture/Tile.....	289

A Beginner's Guide to Gambas – Revised Edition

Drawing with a Drawing object.....	297
Chapter 13 – Error Management.....	300
General Concepts of Error Management.....	300
Error Handling.....	300
Boundary-Related Errors.....	301
Calculation Errors.....	301
Initial and Later States.....	301
Control Flow Errors.....	302
Errors in Handling or Interpreting Data.....	302
Race and Load Conditions.....	303
Platform and Hardware Issues.....	303
Source, Version, and ID Control Errors.....	304
Testing Errors.....	304
Test Plan Reviews.....	304
Gambas Error management.....	305
TRY statement... IF ERROR.....	305
Catch and Finally Statements.....	306
Gambas Event management.....	310
Chapter 14 – Mouse, Keyboard and Bit Operations.....	312
Mouse Operations.....	312
Keyboard Operations.....	316
Bit Operations.....	318
Chapter 15 – MySQL and Gambas.....	326
Blob Class.....	327
Connection Class.....	327
Connection Properties.....	328
The Concept of a Transaction.....	330
Connection Class Methods.....	331
Result Objects.....	335
DB Class.....	336
Database.....	336
DatabaseUser.....	336
Field.....	337
Index.....	337
Table.....	337
The Database Example Program.....	337
Chapter 16 – Global Gambas.....	351
Internationalization.....	351
Localization.....	351
Universal Character Set (UCS).....	351
UTF-8.....	353
File System Safe UCS Transformation Format.....	353
FSS/UTF.....	353
How to translate in Gambas.....	354

Illustration Index

Figure 1: The Gambas Architecture.....	14
Figure 2: The Gambas opening (Welcome) screen.....	15
Figure 3: The Gambas Project Creation Wizard.....	16
Figure 4: Choosing or creating a project directory.....	16
Figure 5-Providing a name and title for the project.....	17
Figure 6: Project creation confirmation dialog.....	17
Figure 7: The Gambas IDE after creating a new project.....	18
Figure 8: The Gambas IDE showing a form in design view and displaying the properties window.....	19
Figure 9: Gambas class page in edit code view.....	20
Figure 10: Gambas IDE File menu.....	20
Figure 11 - The Gambas IDE View Menu.....	21
Figure 12: The Gambas IDE Project Make Menu.....	21
Figure 13: The Gambas Debug Menu.....	21
Figure 14: The Gambas IDE View menu.....	22
Figure 15: The Gambas IDE main menu and toolbar.....	22
Figure 16: Using MMain to develop console applications.....	32
Figure 17: A Division by Zero Error Dialog.....	36
Figure 18: Clarifying the For Next loop structure.....	55
Figure 19: The Gambas Toolbox.....	58
Figure 20: First button code results.....	63
Figure 21: Demonstrating font capabilities.....	65
Figure 22: The dotted line indicates focus for a control.....	69
Figure 23: Layout for the FMain form for the second project.....	70
Figure 24: A partially constructed form with our first four controls.....	71
Figure 25: The Gambas Event Menu.....	72
Figure 26: Adding the FunBtn to our form.....	74
Figure 27: Initial display of our form.....	75
Figure 28: Figure 10: The progress bar when the FunBtn is clicked three times.	75
Figure 29: Third Project final results.....	78
Figure 30: Using HTML to format a text label.....	80
Figure 31: ModifiedTextLabel output using HTML formatting.....	81
Figure 32: Adding a ToolTip to inform the user how to show/hide a control.....	83
Figure 33: Our ComboBox.....	84
Figure 34: The Edit List Property editor.....	85
Figure 35: Formatting aTextLabel with HTML.....	86
Figure 36: Using plus and minus buttons to alter the ComboBox list.....	86
Figure 37: What our ListBox will look like.....	88
Figure 38: The ListBox edit list property editor.....	88
Figure 39: What the example frame we build will look like.....	90
Figure 40: A Panel control containing RadioButtons.....	93
Figure 41: Third Project final results.....	94

A Beginner's Guide to Gambas – Revised Edition

Figure 42: Menu project final result.....	95
Figure 43: The Gambas Menu Editor when it starts.....	96
Figure 44: Editing menu entries in the Menu Editor.....	97
Figure 45: The Gambas Menu Editor.....	100
Figure 46: A formatted text label displaying the color value.....	103
Figure 47: Selecting colors and fonts.....	110
Figure 48: Making a new default font for the TextLabel1 control.....	111
Figure 49: An Information MessageBox.....	112
Figure 50: A checked menu item.....	112
Figure 51: A Question MessageBox.....	113
Figure 52: An Error MessageBox.....	114
Figure 53: A Warning MessageBox.....	115
Figure 54: A Delete MessageBox with three buttons.....	116
Figure 55: The Open File Dialog in Gambas.....	117
Figure 56: The Save File Dialog in Gambas.....	117
Figure 57: The SelectDirectory dialog.....	118
Figure 58: Choosing the Explorer example.....	150
Figure 59: Layout for our ListView example.....	158
Figure 60: Creating a new Icon image in GB.....	163
Figure 61: The Icon Editor Toolbox.....	163
Figure 62: Our square icon image.....	163
Figure 63: Our circle icon image.....	163
Figure 64: Treeview example.....	164
Figure 65: What our GridView will look like.....	171
Figure 66: Our ColumnView example.....	175
Figure 67: Our editable ColumnView example.....	178
Figure 68: Layout project icons.....	181
Figure 69: Form1 design mode showing layout of our controls.....	181
Figure 70: Layout program at program startup.....	184
Figure 71: A TabStrip control.....	185
Figure 72: Tabbed form design layout.....	186
Figure 73: Tab0 layout.....	186
Figure 74: Tab1 layout.....	187
Figure 75: Tab2 ToolButton layout with icons.....	188
Figure 76: Tab3 layout with a ComboBox.....	188
Figure 77: Tab4 layout with a Timer.....	189
Figure 78: The FileOpsQT program at runtime.....	201
Figure 79: The sample application in design mode.....	201
Figure 80: Sample application icons.....	202
Figure 81: Form1.form design mode.....	215
Figure 82: Finished Contacts program.....	252
Figure 83: Toolbar icons used in our sample program.....	255
Figure 84: FMain as seen in design mode.....	257
Figure 85: Designing the search form.....	265
Figure 86: Our finished program running on the Linux Mint desktop.....	271

A Beginner's Guide to Gambas – Revised Edition

Figure 87: gfxDemo FMain layout.....	275
Figure 88: Results of clicking the Text Button.....	278
Figure 89: Results of InvRect button click. Note the tiny black crosshair center screen.....	280
Figure 90: Using the Ellipses button to demonstrate drawing lines and ellipses.....	283
Figure 91: Output after clicking the FillRect button.....	285
Figure 92: Using Draw.Polygon to draw a triangle.....	287
Figure 93: Using Draw.Polyline to draw lines.....	289
Figure 94: Using tiled images as a background on a form.....	297
Figure 95: Loading and displaying graphics in Gambas3.....	299
Figure 96: Error results caught with TRY	306
Figure 97: CATCH test program opening screen.....	308
Figure 98: Div by Zero error caught by CATCH.....	308
Figure 99: The TextLabel updated with error info.....	308
Figure 100: Info message that the error is cleared.....	308
Figure 101: Main screen after error has been cleared.....	309
Figure 102: Default Gambas error dialog.....	309
Figure 103: Our event was raised.....	311
Figure 104: MouseOps program running.....	315
Figure 105: The Keyboard Operations program running.....	318
Figure 106: The BitOps program in design mode.....	320
Figure 107: Our bitOps program running.....	325
Figure 108: The FMain form in design mode.....	338
Figure 109: FRequest Form in design mode.....	339
Figure 110: Choosing the Translate option in Gambas.....	355
Figure 111: The Translate Dialog in Gambas.....	355

Chapter 1 - Introducing Gambas

Gambas was initially created by Benoît Minisini, a resident of the suburbs of Paris. According to Minisini, Gambas is a Basic language with object extensions. The name itself is a play on the words "Gambas almost means Basic" and, according to Minisini, Gambas evolved because of his personal programming experiences with the Microsoft Visual Basic® software product¹. Rather than contend with the horrendous number of bugs and idiosyncrasies he found in that product, he decided to create Gambas.

Gambas is licensed under the GNU Public License² and has taken on a life of its own due to its immense popularity. Gambas runs on most Linux platforms and the current (at the time of this writing) stable version is Release 2.22. Minisini makes it very clear that Gambas is not compatible with Visual Basic and it will never be made compatible. The syntax and internal workings of Gambas are far better and more user friendly. Minisini stated that he took from Visual Basic what he found useful – the Basic language, the development environment, and the ability to quickly and easily make programs with graphical user interfaces.

Minisini disliked the overall poor level of programming common among many Visual Basic programs. This problem may be due to the “*enforced*” use of bad programming practices that have been imposed on developers as a result of the wide array of bugs and strange idiosyncrasies of the proprietary VB language. Gambas was developed to be as coherent, logical and reliable as possible. Because it was developed with an approach designed to enhance programming style and capture the best features the Basic programming language had to offer, the addition of object-based programming has allowed Gambas to become a popular, modern, stable, usable programming environment for Linux developers.

Gambas Architecture

Every program written with Gambas Basic (hereinafter referred to as GB) is comprised of a set of project files. Each file within a project describes a class. The class files are initially compiled and subsequently executed by the Gambas interpreter. This is very similar to how Java works. Gambas is made up of the following programs:

- ✓ A compiler
- ✓ An interpreter
- ✓ An archiver
- ✓ A graphical user interface component
- ✓ A development environment

Figure 1 below is an illustration of the overall architecture of Gambas. In Gambas, a

1 The reader is encouraged to visit <http://gambas.sourceforge.net/index.html> to learn more about the Gambas project.

2 Visit <http://www.gnu.org/licenses/licenses.html#GPL>

project contains class files, forms, modules, and data files. A Gambas project is stored in a single directory. Compiling a project uses an incremental compile method that only requires a re-compilation of the modified classes. Every external reference of a class is resolved dynamically at run time. The Gambas archiver transforms the entire project directory structure into a standalone executable. The Gambas development environment was written with Gambas to demonstrate the fantastic capabilities of the language.

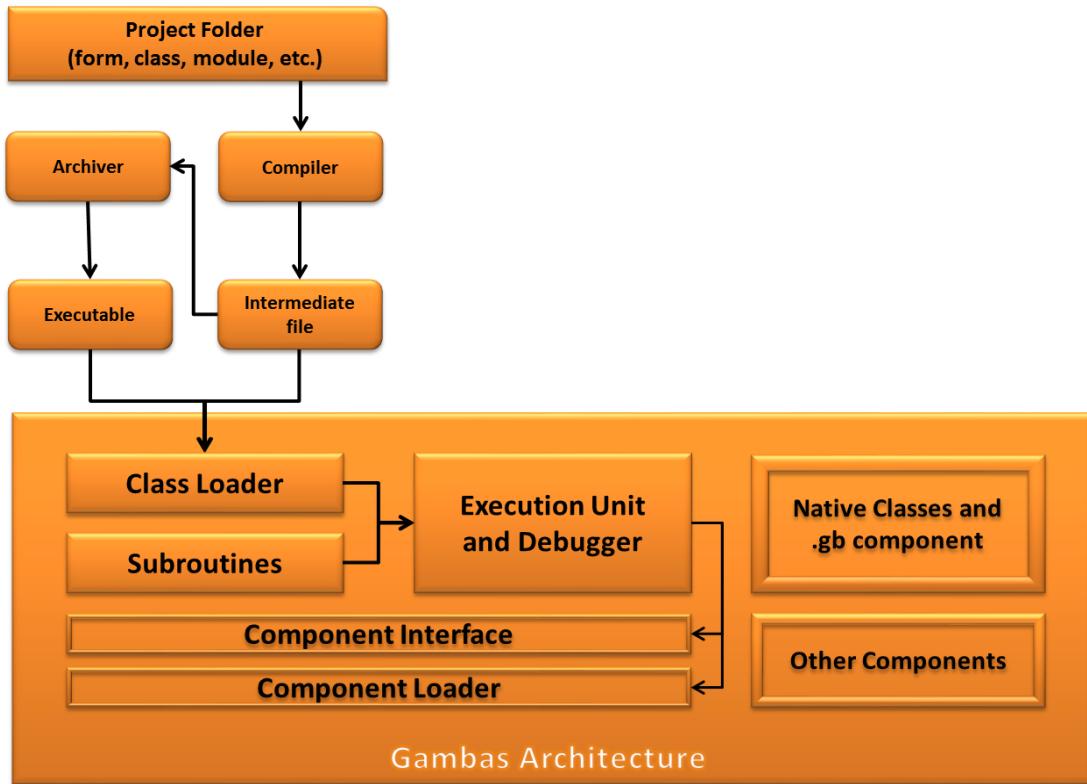


Figure 1: The Gambas Architecture.

Some other features that set Gambas apart from other languages include the fact that Gambas has an extensible component architecture that allows programmers to extend the language. Anyone can write components as shared libraries that dynamically add new native classes to the interpreter. The component architecture is documented online in the **Gambas Wiki encyclopedia**.³ We will cover Gambas components in greater detail later in this book. By default, the Gambas Interpreter is a console-based (text-only) program. The component architecture⁴ is used to support the graphical user interface (GUI) portion of the language. Because the GUI is implemented as a Gambas component, it has the ability to be independent of any specific GUI toolkit. With Gambas, you can write a program and choose which toolkit, such as GTK+⁵, Qt⁶, etc., to use later. The current release of Gambas implements the graphical user interface with the Qt toolkit. The GUI components are derived directly from

³ See the Gambas Wiki at <http://gambas.sourceforge.net/index.html> for more details.

⁴ See <http://gambas.sourceforge.net/architecture.html> for the original rendition of this graphic.

⁵ See <http://www.gtk.org/> for more information on GTK+.

⁶ See <http://www.trolltech.com/products/qt/> for more information on the qt product.

the Qt library. It is recommended that the reader consult with the Qt documentation⁷ to better understand the GUI controls. A GTK+ component with a nearly identical interface to the Qt component has been implemented.



Figure 2: The Gambas opening (Welcome) screen.

Another feature of Gambas that sets it apart from other programming languages is the capability of any window or dialog box to be used like a control. This feature is not directly supported in other programming languages. Also, Gambas projects are easily translatable, into almost any other language. We will demonstrate this feature later in the book.

The Gambas Programming Environment

For now, let's take a quick tour and introduce you to the Gambas programming environment. All of the examples and code written for this book were developed using Gambas version 2.22 running on Ubuntu Linux 10.04⁸ and tested using both Gambas version 2.7 running on Debian⁹ 5.06, and Gambas 3.0 Alpha running on Ubuntu Linux 10.04. They should work on any platform where you can install Gambas successfully. For many Linux users, it is as simple as the click of a button using the software installation features of your specific distribution.

When you first start the Gambas program, you will be presented with an opening screen similar to that found in Figure 2 above. The welcome screen allows you to create a new project, open an existing project, view and select from recent projects, look at examples, or quit. For our quick tour of Gambas, we are going to select the New Project option and create

7 See <http://doc.trolltech.com/4.7/index.html> for more information.

8 See <http://www.ubuntu.com/> for Ubuntu information.

9 See <http://www.debian.org/> for Debian information.

A Beginner's Guide to Gambas – Revised Edition

our first Gambas project. You will see a dialog similar to Figure 3 below appear.

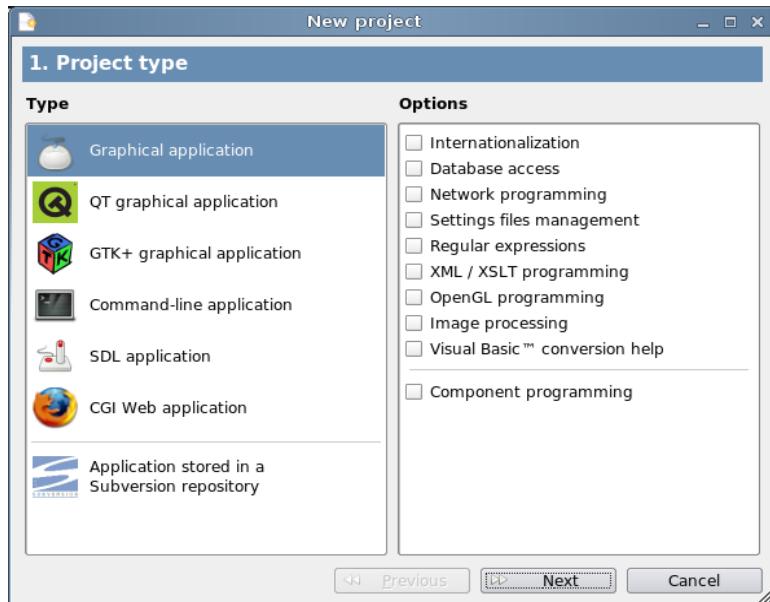


Figure 3: The Gambas Project Creation Wizard.

For our example, choose the first selection, “*Create a graphical project*” and click the **>>Next** button at the bottom of the dialog. The next dialog that you see (shown as Figure 4 below) requires you to select your project directory. You have the ability to create a new folder on your local file system at this step.

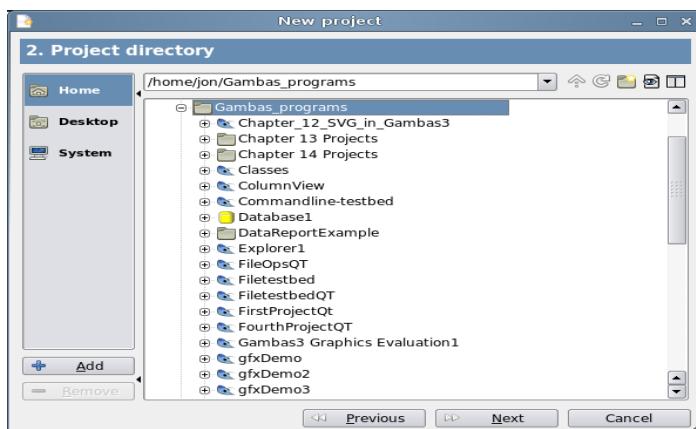


Figure 4: Choosing or creating a project directory.

Click the **>>Next** button at the bottom of the dialog to continue to the next step of the wizard, Project Information. The most important thing to remember about this screen is that it is the last chance you have to back up and make changes before your new project is created.

A Beginner's Guide to Gambas – Revised Edition

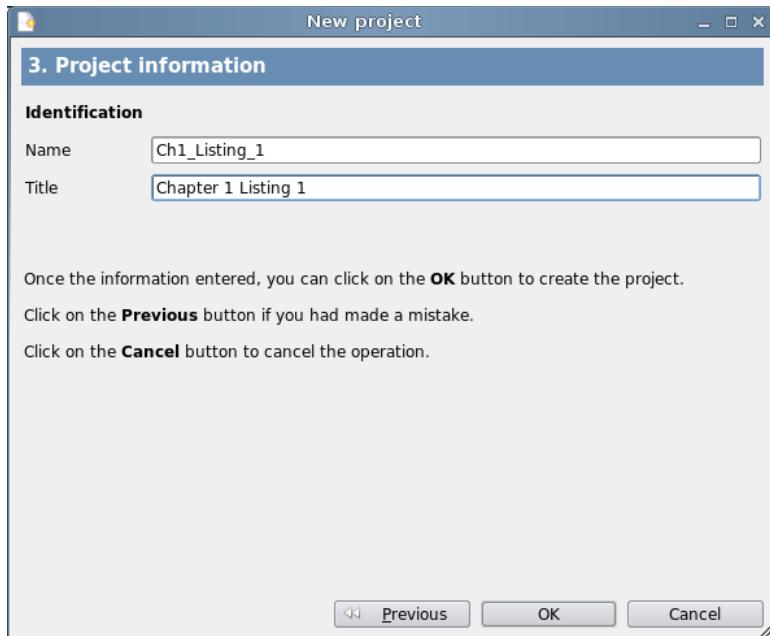


Figure 5-Providing a name and title for the project.

At this section of the wizard, you need to specify the name of the project as it will be stored on disk. In Gambas, a project is self-contained and always stored in its own project folder. The Gambas IDE does not care where you choose to locate your files as it will keep all the required files for your project in the folder you name (e.g., Ch1_Listing_1). The folder name you choose as the name for the project is what will be created and used on your system. The title of the project will be what you specify in the second text input field of the dialog. For now, just fill out the dialog as shown in Figure 5 above and click the **>>Next** button. You should see the following project creation confirmation dialog:



Figure 6: Project creation confirmation dialog.

This dialog is simply a confirmation screen of the choices you have made. Click the **OK** button to continue to the IDE and your new project. Once you have clicked on the “**OK**” button to finish this wizard, Gambas will present you with the Gambas integrated development environment (IDE) as shown in Figure 7 below.

A Beginner's Guide to Gambas – Revised Edition

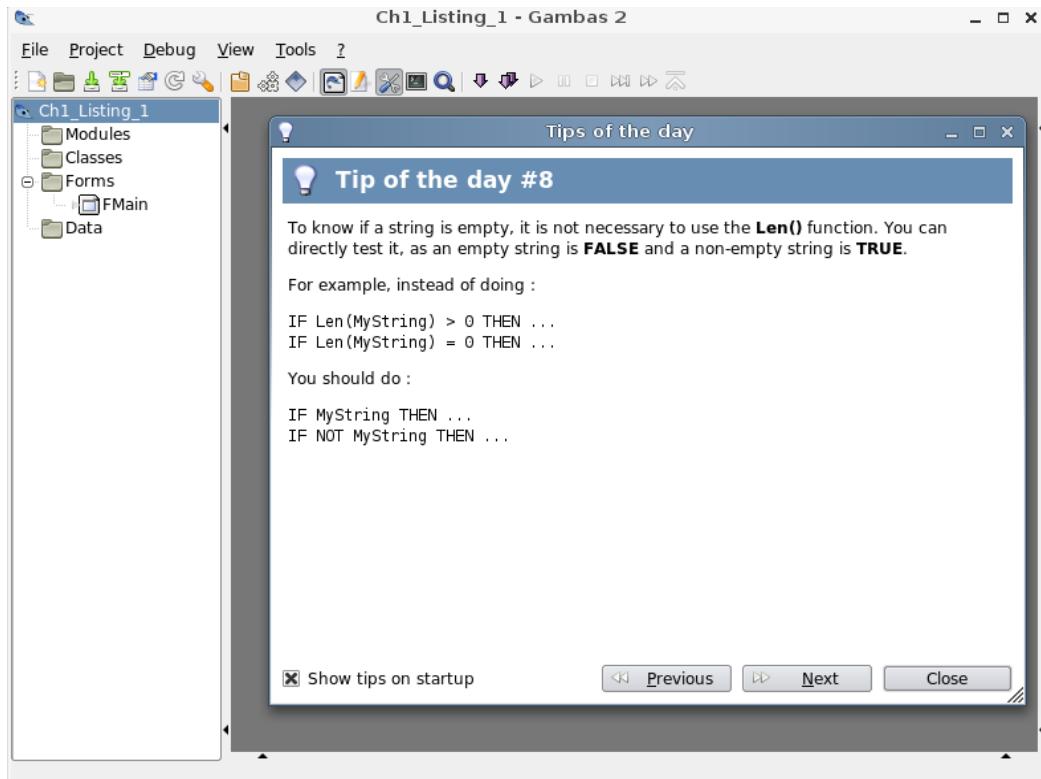


Figure 7: The Gambas IDE after creating a new project.

Gambas IDE Components

The **Project Explorer** (seen on the left side of Figure 8 below) is the main navigation tool used in the Gambas IDE. It shows you a TreeView of the types of files found within your project (*i.e., class files, forms, modules, and other types of files such as data and picture or icon files*). The Project Explorer allows you to perform most Gambas project management operations, such as opening and saving projects, building executables, running your program or debugging it, showing or hiding various Gambas dialogs, etc. From the TreeView in the Project Explorer, you will see the items listed as follows:

- ✓ Modules
- ✓ Classes
- ✓ Forms
- ✓ Data

Modules display the modules you've written for your project. Modules are simply sets of subroutines and functions to be used anywhere in your program. Unlike classes, you can't make objects out of them at run time and they have no event handlers.

Classes lists the class files you've created for your project. Classes are basically templates that can be used to make objects out of at run time, with code to define properties,

A Beginner's Guide to Gambas – Revised Edition

methods and event handlers for each object you create.

Forms lists the various forms you create for your project. Forms are the windows the user actually interacts with.

Data lists the other files in your project. These can include any other kind of file used to build your project, such as graphic files, icons, bitmaps, text or HTML files, and even media files. At the bottom of the Project Explorer, you will find the **Status Bar** which is used to indicate what Gambas is currently doing.

When you start to develop a project in Gambas, you will usually want to begin with a main form. The main form is where the program startup and initialization will occur and it is usually the first thing your user will see when they execute (or run) the application you have built. By default, this form is named FMain. This is the form you will add controls to and specify what actions are to be taken when the user interacts with those controls. Such interactions between the user and the GUI are referred to as events.

The controls Gambas uses are found in the Toolbox found on the lower right corner of Figure 8 below. You can change the appearance and behavior of the controls by setting the properties for each control. Properties can be seen and changed from within the properties window shown on the right of side of Figure 8 below. You can see a form under construction (with no controls added at this point) displayed as the grayed rectangle center-screen.

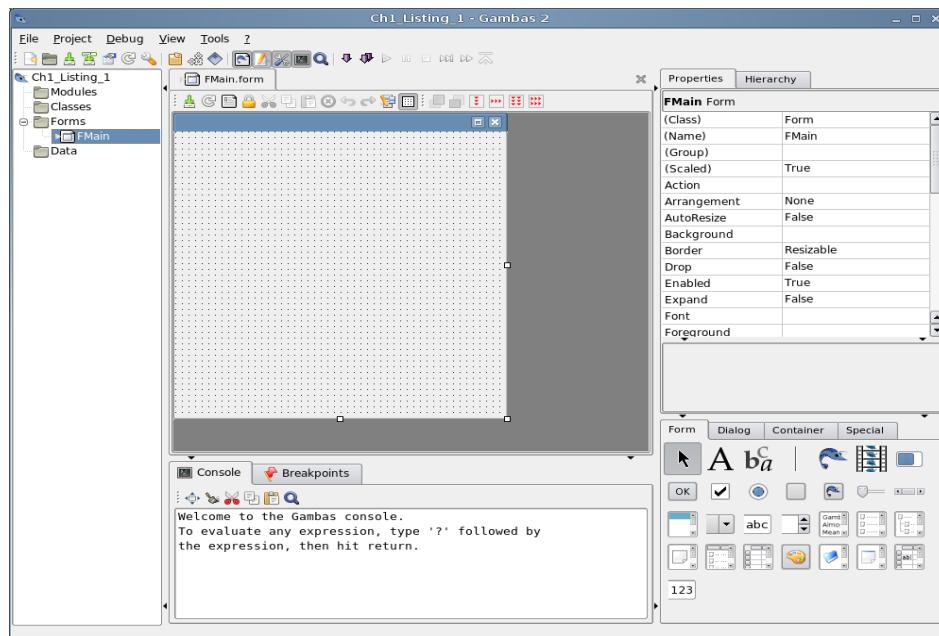
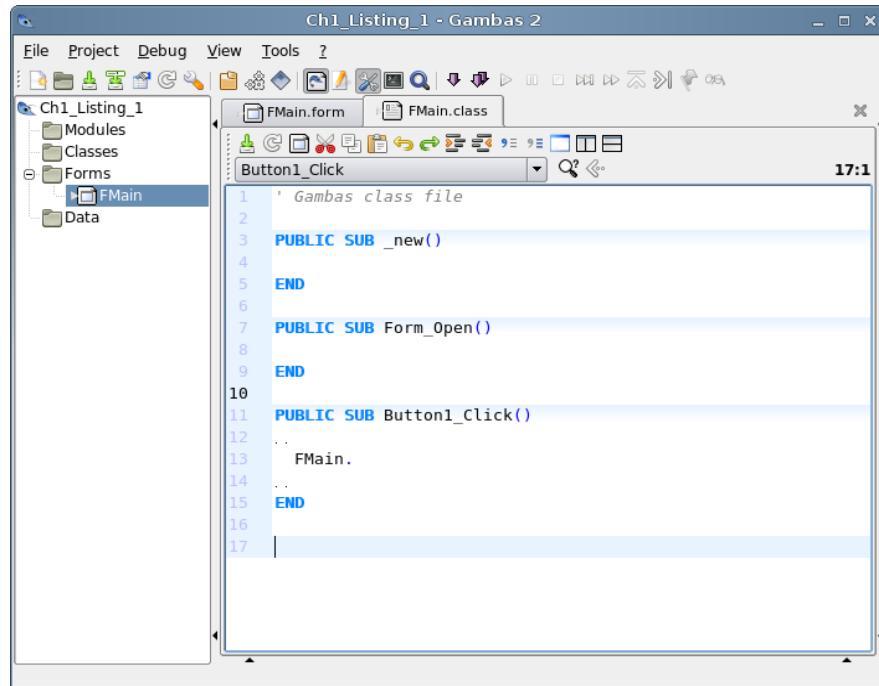


Figure 8: The Gambas IDE showing a form in design view and displaying the properties window.

A class page is being coded in Figure 9 below. You will work with modules, classes and

A Beginner's Guide to Gambas – Revised Edition

forms as tabbed pages in the IDE window, similar to those shown in these examples. From the figure below, you can see that the syntax of Gambas is very similar to that of other visual basic languages, but not enough to be interchangeably used.



```
1 ' Gambas class file
2
3 PUBLIC SUB _new()
4
5 END
6
7 PUBLIC SUB Form_Open()
8
9 END
10 PUBLIC SUB Button1_Click()
11 ...
12 FMain.
13 ...
14 END
15
16
17 |
```

Figure 9: Gambas class page in edit code view.

The Gambas development environment provides the features such as syntax highlighting of Gambas code, HTML and CSS files, automatic completion of keywords, a GUI form editor and integrated debugger, an icon editor, and a string translator. Some other features of this code editing environment include color-coding for keywords, line numbering (useful for debugging), and structured indentation.

Now, let's consider the menus and buttons found at the top-left corner of the Project Explorer. The menus control all the main Gambas management tasks. The File menu (Figure 10) will allow you to open a project, save a project, create a new project, open some Gambas example projects, or quit using Gambas.

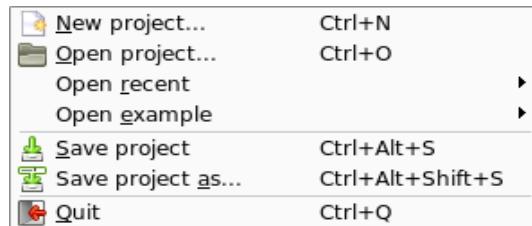


Figure 10: Gambas IDE File menu.

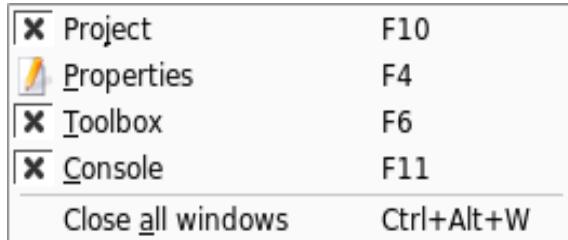


Figure 11 - The Gambas IDE View Menu.

The Project Menu is where program compilation occurs. You can create the program executable, make a source archive or an installation package from this menu by choosing the Make menu item and selecting the appropriate entry. This menu also provides housekeeping functions and opens a dialog with a range of options, including translating your program into another language.



Figure 12: The Gambas IDE Project Make Menu.

The Debug Menu lets you run your program inside the IDE, and offers controls to isolate problems with your programming code. When your executable is running, you can pause it, stop it, step forward through the code by executable statement, or finish the executing process and return to programming. If you have set breakpoints in the code edit window, you can also clear them using this menu.



Figure 13: The Gambas Debug Menu.



Figure 14: The Gambas IDE View menu.

The View menu will allow you to bring up the Project Explorer, the Properties pane, or view the ToolBox pane. You can open a console from here. When the Properties window is open (as shown in Figure 8, on the right-hand side of the picture), the Hierarchy option appears as a page tab, and this page shows you your project's class hierarchy. Finally, from this menu, you can close all of the windows that are currently open in the IDE.

Figure 15 below shows you the File menu and toolbar. The **ToolBar** buttons provide single-click access to the most common menu items. Hovering your mouse cursor over one of the buttons will display a **ToolTip** that will tell you what menu action that particular button will perform.



Figure 15: The Gambas IDE main menu and toolbar.

From the Project Explorer **TreeView** you can double click on a form and it will pop up for you to edit (as shown in Figure 9 above). Editing forms is simply a matter of selecting what type of control you'd like to place on the form and then using the mouse to resize (draw) and position it on your form.

Right clicking on the form or any of its children (controls) will show you a pop-up menu that will allow you to perform operations on the control, edit its properties or delete it. The currently selected control is indicated by four black squares called handles. Clicking on a handle and using the mouse to drag the control where you want it will allow you to move or resize the control. Double clicking on a control will bring up the code editor window and display any existing event handler for the control or display a default event handler if there have been none specified.

The Gambas Code Editor enables you to write code to handle events for the controls you've placed on your form. In the toolbox window, the Selection Tool is the only item in the toolbox that isn't actually a control. The selection tool simply gives you control of the default mouse pointer. You use this pointer to select controls and perform positioning and resizing operations on forms and their associated controls.

A Beginner's Guide to Gambas – Revised Edition

From the File Menu at the top of the IDE, choose the Quit option and save this project. When we reopen it, all your work will be saved. In the next chapter, we will begin to cover the essentials you need to know in order to program effectively with Gambas.

We will come back to this project to develop a program that uses the GUI after we learn a bit more about the GB coding environment, the primary language concepts that are required to use GB, and some basics about data-types and variables.



Chapter 2 – Language Concepts

In this chapter, we will begin to learn the basic concepts needed to master the Gambas programming language. The topics we will cover in this chapter include learning about Gambas data-types, constants, and variables and how to declare and assign values to those constants and variables. We will learn about the basic arithmetic operators, comparison operators, and string operators. When we start writing program code, the code will be organized in “code blocks” that will look something like this:

```
PUBLIC SUB Main()  
    DIM variable as variable type  
    DIM another variable as another variable type  
    statement...expression  
    more like it...etc.  
END
```

We will learn more about code blocks later, but let's look at the keywords in bold.

- ✓ **PUBLIC** lets other parts of your program (other "classes") access this code block.
- ✓ The keyword **SUB** comes from the word “subroutine”, and in Gambas it is a keyword to declare a procedure -- some body of code written to do a specific job. The **SUB** keyword is synonymous with the **PROCEDURE** keyword, and these two can be used in code interchangeably.
- ✓ Note that a **SUB** has some similarity to a **FUNCTION**, but with one big difference: a function has limited scope. The only job of a **FUNCTION** is to accept parameters from some part of a program, process the parameters and return a value to that same place in the program where the **FUNCTION** was called. In Gambas, procedures and functions together are referred to as “methods”.
- ✓ In this particular case, **Main()** signals to Gambas that the instructions inside this code block will be the instructions that create the initial conditions of the running program. Therefore, the page that this code appears on will be the “startup class” for your program.
- ✓ The keyword **DIM** comes from the word “dimension”. Dimensioning a variable means that the program sets aside a specific block of memory, of a specific size, to hold the data represented by that variable.
- ✓ **END** marks the end of this procedure.

So, looking at the entire first line we can see that this line describes a "method" (a **SUB**) that will allow Gambas to use the "class file" or "module file" containing this code block as the startup class. This will be the first section of code that Gambas will look for, when you run your program. Between the first line and **END**, you can use more Gambas code to describe the other events that you want to happen when the program starts.

Gambas Variables, Data-types and Constants

Variables must be defined at the beginning of a class, method or function. Variable declarations can either be *local* to a procedure or function or they can be declared *global* to a class. A global variable is accessible everywhere **in the class** it is declared. The format of a global variable declaration in Gambas takes the following general form:

[STATIC](PUBLIC|PRIVATE) Identifier [Array declaration] AS [NEW] Data-type

If the **PUBLIC** keyword is specified, it is also accessible to the other classes that have any reference to an object of that class. If the **PRIVATE** keyword is specified, it is not accessible outside the class **in which it was defined**. If the **STATIC** keyword is specified, the same variable will be shared with every object of the class where it is declared. If the **NEW** keyword is specified, the variable is initialized with (*i.e.*, *instantiated with*) a new instance of the class using the data-type specified. For local variable declarations, the format is like this:

DIM Identifier AS Datatype

This will declare a local variable in a procedure or function. This variable is only accessible to the procedure or function where it is declared. An example of several local declarations is shown below:

**DIM iValue AS INTEGER
DIM stMyName AS STRING
DIM fMyMatrix[3, 3] AS FLOAT
DIM oMyObject AS OBJECT**

There are currently eleven basic data-types a programmer can use to write program code in Gambas. These data-types include the following:

Boolean	Byte	Short	Pointer
Integer	Long	Float	Single
Date	String	Variant	Object

Boolean data-types can contain only a single value, either TRUE or FALSE. TRUE is defined as 1 and FALSE is defined as 0. The declaration of a Boolean variable occupies 1 byte of memory and the default value is FALSE. An example of a variable declaration for a Boolean variable looks like this:

STATIC PRIVATE bGrid AS Boolean

Typically, a programmer would use a Boolean data-type when the only values for the

variable would be yes/no, TRUE/FALSE, 1/0, etc. If the values used in your program could be anything else, Boolean would be an inappropriate selection for the variable data-type. Another thing to note in the variable declaration above is the placement of the lowercase letter b in front of the variable name.

Good programming convention encourages this practice, known as the Hungarian Notation¹⁰, as it allows the programmers to know what data-type the variable is by simply knowing that the 'b' stands for Boolean. What happens when a programmer wants to use a Byte data-type instead of a Boolean data-type? Typically, a second letter is added to the variable declaration so rather than using 'b' in front of the variable name the programmer would use 'by' as below:

STATIC PRIVATE bySomething AS Byte

The letters 'ar' 's', 'i', 'f', 'd', 'st', 'v' and 'o' are commonly used notations when declaring variables while programming in Gambas. It is good programming practice to force yourself to adhere to this technique so that others can pick up your code and use it without having to search around to find the data-type for each variable encountered. Some programmers even use more than the first letter(s) to name their variables. For example, they would code **IntMyNumber** or **ByteSomething**.

The **Byte data-type** can contain values from 0 to 255 and occupies one byte of memory when declared. The default value when a Byte data-type is declared is 0. If you are certain your values will not exceed 255 then this data-type is appropriate for use. If it is possible that the values assigned to this variable would exceed 255, then the likely result would be a program crash at run time.

It is better to use the **Short data-type** or an **Integer data-type** for such situations. For a Short data-type, values can range from -32768 to +32767. Short data-types occupy two bytes of memory and default to a zero value when declared. Integer data-types occupy twice as much memory, taking up four bytes. The range of values you can use with the Integer data-type is from -2,147,483,684 to +2,147,483,647, and the Integer data-type can accept numbers written in decimal, hexadecimal or binary.

The **Long data-type** is an eight-byte integer value, that accepts integers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807. Examples of Short and Integer data-type variable declarations are:

STATIC PUBLIC sSomeShort AS Short **STATIC PUBLIC iSomeInteger AS Integer**

When integers will not work for your purposes, Gambas provides you with the **Float data-type**. This data-type allows you to use floating point numbers for your program. Float data-types are like Double data-types used in C and VB. The range of values for float data-

¹⁰ Hungarian Notation (HN) is a naming convention originated by Charles Simonyi of Microsoft. It was first presented in his thesis and is widely used throughout the source code of the Windows operating system, among other places.

type variables is from -1.79769313486232E308 to -4.94065645841247E-324 for negative values and from 4.94065645841247E-324 to 1.79769313486232E308 for positive values. Float variables occupy eight bytes of memory and default to a zero value when declared. A sample declaration for a float would be as follows:

```
DIM fRadius AS Float
```

Gambas also provides the Single data-type, which holds a 4-byte floating point value between -1.7014118E-38 and 1.7014118E+38.

The last numeric data-type we have in Gambas is the **Date data-type**. Date variables also take up eight bytes of memory. The date portion of the date is stored in a four byte integer and the time portion is stored in a four byte integer. It is stored as **[Year, Month, Day][, Hours, Minutes, Seconds]** and is usually used with the Gambas built-in Date and Time functions, which we will explain about later in this book. The date data-type defaults to a NULL value when initialized. Here is how to declare a Date data-type:

```
DIM ddate AS Date  
DIM dtime AS Date
```

Strings, Variants and Objects are the non-numeric data-types supported in Gambas. A **String data-type** is a series of zero or more characters that are treated as a single entity. Strings can contain alphanumeric data. Alphanumeric means that the data can contain any combination of letters and integers or special characters such as \$%^&*.

Strings, when declared take four bytes of memory. This means the maximum size for a string is 4 bytes * 8 bits per byte, or 32 bits squared (1,024 bytes). String variables default to a NULL value when declared. Declare a string variable just like you would any other variable:

```
STATIC PUBLIC stSomeString AS String
```

The **Variant data-type** is used when you do not know what kind of data-type the variable will be receiving. For example, if reading data from a file, you could read an integer, a string, a single character, floating point numbers, etc. To ensure the data is placed in a variable without causing a program crash, the variant data-type is used.

You can then test the variant data using some built-in functions of Gambas to determine the data-type or you can just convert the data to the data-type you need using a conversion function. We will demonstrate this later in the book. For now, it is only important that you understand that variant data-types exist and that they are used when you are not sure of the type of data the variable will hold.

The **Object data-type** is a special data-type that holds and references objects such as controls and forms. Later, when we begin to discuss OO programming, we will cover the use of object data-types in greater detail.

The native **Pointer** datatype represents an address in memory. It can be declared like this:

DIM Var AS Pointer

This is equivalent to the [Integer](#) datatype on 32 bits systems. In the 64-bit version of [Gambas](#), a Pointer is equivalent to the [Long](#) datatype. In Gambas 3, Pointer is a true data type and you cannot replace a pointer with an integer number. The table shown on the following page is presented as a convenient reference. Now that you have learned a little about all of the different types of data that Gambas supports, we will start to look at what you can do with those.

When using your variables in Gambas programs, they can be represented by data that changes (*e.g., it is a variable*) or they can be represented by data that remains constant throughout the program. This type of data is known as a **Constant** in Gambas. Gambas constants is used to represent a NULL object reference, a zero length string, a NULL date, or an uninitialized variant. Examples of constants include the values NULL, TRUE and FALSE.

Gambas data-types

Name	Description	Memory size	Default
Boolean	True or False	1 byte	FALSE
Byte	0 ... 255	1 byte	0
Short	-32768 ... +32767	2 bytes	0
Integer	-2147483648 ... +2147483647 (base10, hex and binary)	4 bytes	0
Long	-9,223,372,036,854,775,808... +9,223,372,036,854,775,807	8 bytes	0
Float	Similar to the double data-type in C	8 bytes	0
Single	-1.7014118E-38... 1.7014118E+38	4 bytes	0
Date	Date/time, each stored in a 4 byte integer.	8 bytes	NULL
String	A reference to a variable length string.	4 bytes	NULL
Variant	Can consist of any data-type.	12 bytes	NULL
Object	An indirect reference to an object.	4 bytes	NULL
Pointer	A memory address.	4 bytes on 32 bit systems 8 bytes on 64 bit systems	0

To declare a constant in Gambas use the following format:

(PUBLIC | PRIVATE) CONST Identifier AS Datatype = value

This declares a class global constant. This constant is accessible everywhere in the class it is declared. If the **PUBLIC** keyword is specified, it is also accessible to the other classes

A Beginner's Guide to Gambas – Revised Edition

having a reference to an object of this class. Constant values must be Boolean, integers, floating point or string data-types. Here are some examples of constant declarations:

```
PUBLIC CONST MAX_FILE AS Integer = 30
```

```
PRIVATE CONST MAGIC_HEADER AS String = "# Gambas form file"
```

The built-in constants you would use in Gambas are listed in the table below:

Gambas Constants

Constant	Example
The TRUE value.	TRUE
The FALSE value.	FALSE
Integer numbers.	0, 562, 17, -32769
Hexadecimal short signed integers.	&H100F3, &HF0FF, &FFFF
Hexadecimal signed integers.	&H1ABF332E, &1CBF302E
Hexadecimal unsigned integers.	&H80Ao&, &HFCFF&
Binary integers.	&X1010111101, %101000011
Floating point numbers.	1.1110, -5.3419E+4
String constants.	"Hello, Gambas World!"
String constants to be translated.	("This is very, very cool")
NULL constant / void string.	NULL

String constants can also contain the following escape characters:

Escape character	ASCII equivalent
\n	CHR\$(13)
\r	CHR\$(10)
\t	CHR\$(9)
\"	Double quote
\\\	Backslash
\xx	CHR\$(&Hxx)

You can write a string constant in several successive parts. For example, “My son” “ is “sixteen” is seen by Gambas as “My son is sixteen”.

Variable Assignment

A programmer can assign any value to a variable in Gambas by using the following general format:

Variable = Expression

This *assignment statement* assigns the value of an expression to one of the following elements:

- ✓ A local variable
- ✓ A function parameter
- ✓ A global (class) variable
- ✓ An array element
- ✓ A public object variable or property

Here are some example of variable assignments:

```
iMyVal = 1984  
stMyName = "Orwell"  
fMyNum = 123.45
```

Assignment Using The WITH Statement

This statement is most commonly used to set properties for controls. The expression that exists between the **WITH keyword** and the **END WITH** instruction is used. The expression will begin with dot notation, i.e., .Text could be used. WITH assigns the dotted expression on the left of the equal sign the value found on the right side of the equal sign. *Expression* must be an object. Here is a sample of how the WITH structure looks:

```
WITH Expression  
.object = "something"  
END WITH
```

As an example, the code below is code equivalent to **hButton.Text = "Exit"**

```
WITH hButton  
.Text = "Exit"  
END WITH
```

Operators and Expressions

Now that we know how to declare variables and constants and how to assign values to these variables and constants, lets take a look at the operations that can be performed with them. We will begin with arithmetic operators then take a look at comparison operators and string operators.

Arithmetic Operators

All of the basic arithmetic operations are supported in Gambas. These operators include addition, subtraction, multiplication, and division. The standard symbols for these operations are '+', '-', '*', and '/'. For example, **Number + Number** will add two numbers. When a value or variable is preceded by a minus sign, **-222**, for example, Gambas computes the opposite sign of that number. The value Zero is the opposite of itself.

Now, we will start to write some Gambas code using the Gambas terminal application feature. The console window will display our output so lets use Gambas to experiment with the operators as we learn about them.

Comparison operators

Comparison of two variables requires finding answers to questions like “*does x equal y*” or “*is a less than b*”. The following comparisons are supported in Gambas:

Operator	Meaning	Example
=	Is equal to	IF a = b THEN ...
<>	Is not equal	IF a <> c THEN ...
<	Is less than	IF a < d THEN ...
>	Is greater than	IF a > e THEN ...
<=	Is less than or equal to	IF a <= f THEN ...
>=	Is greater than or equal to	IF a >= g THEN ...

Let's Start Coding Gambas

Now that we know about data-types and variable assignments, lets get our feet wet with Gambas. Start Gambas and from the File Menu select New Project. As you go through the New Project Wizard, select a Command-line application and click **>>Next**. Place this project in a directory called *CommandlineTest* and name it *CommandlineTest*. Don't worry about any of the other options. Just click **>>Next** until the wizard completes.

Once the IDE appears with your new project we will need to create a startup class in order to run our code. In Gambas, a command line application is based on a special class called a “module”, which is a “STATIC” class (we will talk more about the STATIC declaration later).

From the Project Explorer find the TreeView item called **Modules** and double-click the mouse on the icon labeled “MMain”. This is always your default module for a command line application. The code window will appear similar to the figure shown below. Inside the window, you should look something very similar to the figure left.

A Beginner's Guide to Gambas – Revised Edition

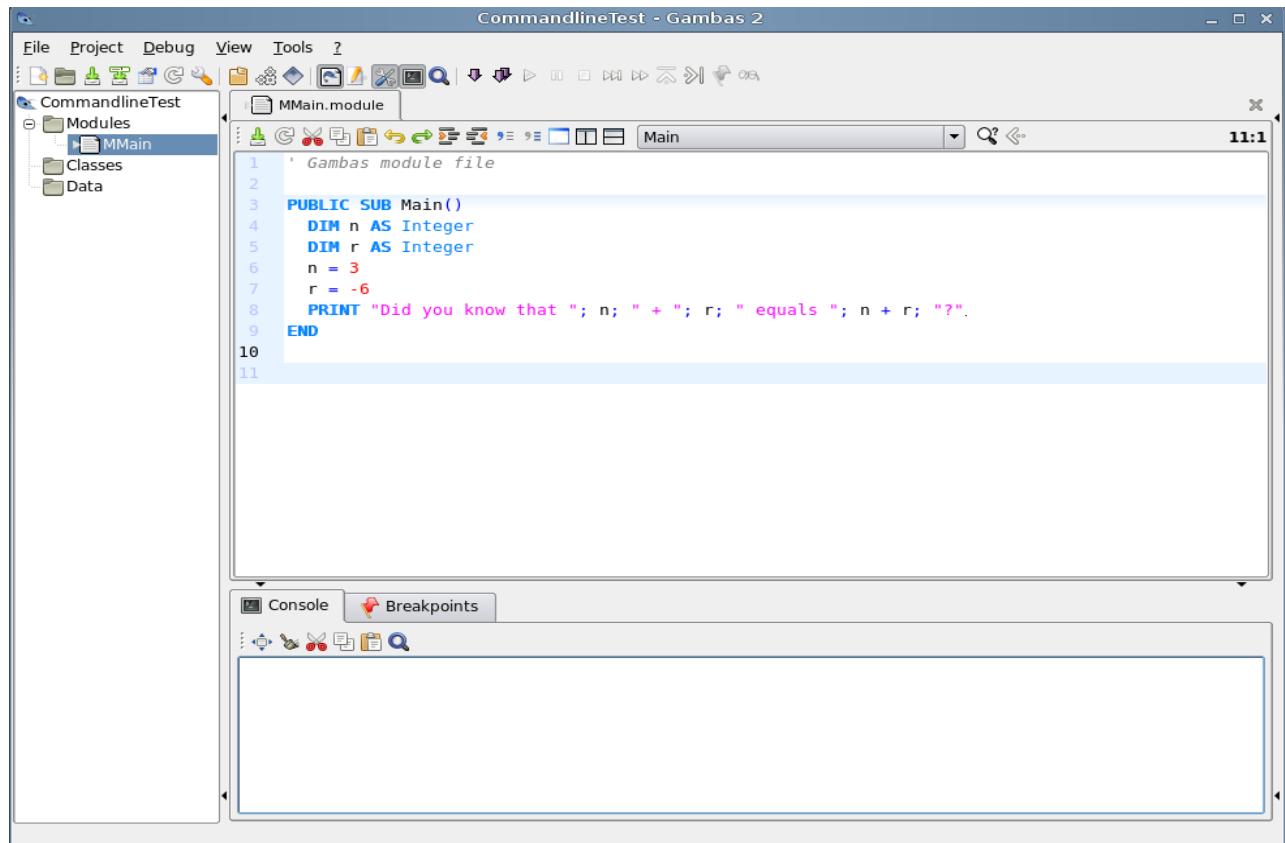


Figure 16: Using MMain to develop console applications.

In Gambas, when we start writing programs, the code will be organized in “code blocks” that will likely look something like this:

```
PUBLIC SUB Main()
    DIM variable as variable type
    DIM another variable as another variable type
    statement...expression
    more like it...etc.
END
```

We will learn more about code blocks later in the book, but let's look at some of the more significant keywords shown above in bold letters, starting with **PUBLIC** – it lets other parts of your program (other “classes”) access this code block. **SUB** comes from the word “subroutine”, and in Gambas it is a keyword to declare a procedure – some body of code written to do a specific job. The **SUB** keyword is synonymous with the **PROCEDURE** keyword, and these two can be used in code interchangeably. Note that a SUB has some similarity to a **FUNCTION**, but with one big difference: a function has limited scope – its only job is to accept parameters from some part of a program, process the parameters and return a value to that same place in the program where the **FUNCTION** was called. In

A Beginner's Guide to Gambas – Revised Edition

Gambas, procedures and functions together are referred to as “methods”. In this particular case, **Main()** signals to Gambas that the instructions inside this code block will be the instructions that create the initial conditions of the running program.

The keyword **DIM** comes from the word “dimension”. Dimensioning a variable means that the program sets aside a specific block of memory, of a specific size, to hold the data represented by that variable. **END** marks the end of this procedure. So, looking at the entire first line of code, we can see that this line describes a method (i.e., a **SUB**) that will allow Gambas to use the class or module file containing this code block as the startup class. This will be the first section of code that Gambas will look for, when you run your program. Between the first line and **END**, you will use Gambas code to describe the other events that you want to happen when the program starts. Now, let's take a look at some other important Gambas keywords you should know a bit more about before we proceed.

END, RETURN and QUIT Statements

The **END** keyword indicates the end of a procedure or a function. There are differences from VB when using **END**. In VB, the **End** command closes all forms and files and terminates the program. In Gambas, the **END** command works more like VB's **End Function** combined with VB's **End Sub**. It closes the function or subroutine. For similar functionality to VB's **End** command, use the **QUIT** command. It ends the program immediately. All windows are closed, and everything is freed up in memory as cleanly as possible.

In Gambas, when you wish to exit a routine, you can use the **RETURN** command. Usually, **RETURN** is used to return a value to the calling procedure. The format of **RETURN** is:

RETURN [Expression]

When Gambas executes the **RETURN** command, it quits a procedure or a function and completes its work by returning the value of *Expression*. Now, enter the following code after the '**Gambas module file**' line (note that comments in Gambas start with the '*[aka., a tick mark]*') and exist between the **PUBLIC SUB Main()** and **END** statements. Once you have entered the code shown below in the Gambas code window, click the  button on the ToolBar to execute your program. Here is the code you want to try first:

```
PUBLIC SUB Main()
DIM n AS Integer
DIM r AS Integer
n = 3
r = -6
PRINT "Did you know that "; n; " + "; r; " equals "; n + r; "?"
END
```

If all goes well, you will see the console window respond with the following:

Did you know that $3 + -6$ equals 3?

Don't worry about the syntax of the **PRINT** statement or the use of the keyword **DIM** used to declare our variables for now. We will cover these keywords later in the book.

To subtract values, use the format **Number - Number** and Gambas will subtract the second number from the first.

```
PUBLIC SUB Main()
    DIM N AS Integer
    DIM R AS Integer
    N = 8
    R = 5
    PRINT "==> "; N-R;
END
```

The console will respond with the following:

==> 3

To multiply numbers, we use the format of **Number * Number** and Gambas will multiply the two numbers. Here is another console example to try:

```
PUBLIC SUB Main()
    DIM N AS Integer
    DIM R AS Integer
    N = 8
    R = 5
    PRINT "==> "; N * R;
END
```

The console will respond with the following:

==> 40

Division is no different than multiplication. Use the format of **Number / Number** to have Gambas divide two numbers. A division by zero error will occur if the value of the number *to the right* of the slash is zero. Try this console example:

```
PUBLIC SUB Main()
    DIM N AS Integer
```

```
DIM R AS Integer  
N = 9  
R = 3  
PRINT "====> "; N / R;  
END
```

The console will respond with the following:

```
====> 3
```

Now try using the \ to divide instead of the / character:

```
PUBLIC SUB Main()  
DIM N AS Integer  
DIM R AS Integer  
N = 9  
R = 5  
PRINT "====> "; N \ R;  
END
```

The console will respond with the quotient:

```
====> 1
```

This operation is called “integer division”. If you use a backslash to divide numbers, i.e., **Number \ Number** Gambas computes the **quotient** of the two numbers, without the remainder. A division by zero error will occur if the value of the number to the right of the backslash is zero. **A \ B** is the equivalent of **INT(A/B)**. To get the remainder, we can use the built-in **MOD** function like this:

```
PUBLIC SUB Main()  
DIM N AS Integer  
DIM R AS Integer  
N = 9  
R = 5  
PRINT "====> "; N \ R; " and the remainder is: "; N MOD R;  
END
```

The console responds with:

```
====> 1 and the remainder is: 4
```

Using **Number MOD Number** computes the **remainder** of the quotient of the two numbers. A division by zero error will occur if the value of the number to the right of the

MOD operator is zero. Finally, we can test the Division by Zero error by typing this example:

```
PUBLIC SUB Main()
    DIM N AS Integer
    DIM R AS Integer
    N = 9
    R = 0
    PRINT "====> "; N / R;
END
```

Gambas will respond with the following:

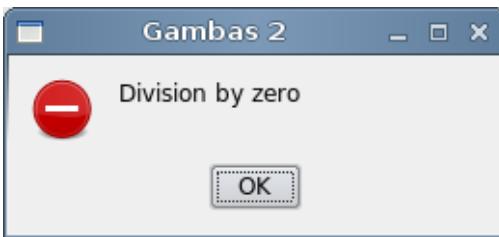


Figure 17: A Division by Zero Error Dialog.

NOTE: Click the Stop button when you see this dialog appear.

In order to raise a number to a given power (exponent), use the format of **Number ^ Power** and Gambas raises Number to the power the **Power** operator specified. Try this:

```
PUBLIC SUB Main()
    DIM N AS Integer
    DIM R AS Integer
    N = 2
    R = 3
    PRINT "====> "; N ^ R;
END
```

The console will respond with the following:

====> 8

Gambas also has the ability to support logical arithmetic operations¹¹. Using the format of **Number AND Number** instructs Gambas to use the **AND** operator (**AND** stating “both are true”) to compute the mathematical **AND** of the binary values of both of the numbers. Try this:

¹¹ In Gambas2, it will allow the use of concatenated operators. For example, `a += 2` or `B /= 4` will work just as if you were programming in C or C++.

```
PUBLIC SUB Main()
DIM N AS Integer
DIM R AS Integer
N = 0
R = 0
PRINT "=> ";N AND R;" is the AND result of ";N;" and ";R
R = 1
PRINT "=> ";N AND R;" is the AND result of ";N;" and ";R
N = 1
PRINT "=> ";N AND R;" is the AND result of ";N;" and ";R
END
```

The console window responds with:

```
=> 0 is the AND result of 0 and 0
=> 0 is the AND result of 0 and 1
=> 1 is the AND result of 1 and 1
```

The AND operator can only be true (1) if both terms are true. Likewise, **Number OR Number** used the **OR** operator (**OR** stating “at least one is true”) and computes the mathematical **OR** of the binary value of the two numbers.

```
PUBLIC SUB Main()
DIM N AS Integer
DIM R AS Integer
N = 0
R = 0
PRINT "=> ";N OR R;" is the OR result of ";N;" OR ";R
R = 1
PRINT "=> ";N OR R;" is the OR result of ";N;" OR ";R
N = 1
PRINT "=> ";N OR R;" is the OR result of ";N;" OR ";R
END
```

The console window responds with:

```
=> 0 is the OR result of 0 OR 0
=> 1 is the OR result of 0 OR 1
=> 1 is the OR result of 1 OR 1
```

The OR operator can only be true (1) if one or both terms are true.

A Beginner's Guide to Gambas – Revised Edition

Number XOR Number uses the **XOR** operator (**XOR**, “**Exclusive OR**”, stating that “one and only one is true”) and computes the mathematical exclusive OR of the binary value of the two numbers.

```
PUBLIC SUB Main()
DIM N AS Integer
DIM R AS Integer

N = 0
R = 0
PRINT "=> ";N XOR R;" is the XOR result of "; N; " XOR "; R
R = 1
PRINT "=> ";N XOR R;" is the XOR result of "; N; " XOR "; R
N = 1
PRINT "=> ";N XOR R;" is the XOR result of "; N; " XOR "; R
END
```

The console window responds with:

```
=> 0 is the XOR result of 0 XOR 0
=> 1 is the XOR result of 0 XOR 1
=> 0 is the XOR result of 1 XOR 1
```

The **XOR** operator can only be true (1) if one and only one term is true. Additionally, the following operators manipulate one or more numeric values and return a numeric value:

DEC | INC | LIKE | NOT

The **DEC** operator is used to decrement a value by one. The **INC** operator will increment the value by one. The variable can be any target of an assignment but must be a numeric value.

```
PUBLIC SUB Main()
DIM N AS Integer
DIM R AS Integer
N = 5
R = 5
DEC N
INC R
PRINT "====> "; N; " | "; R;
END
```

The console will respond with the following:

```
==> 4 | 6
```

LIKE is used to perform a Boolean comparison of a string. It takes the format of **String LIKE Pattern** and it returns TRUE if the String matches the Pattern. The pattern can contain the following pattern-matching characters :

* Match *any* number of any type of character. If you only want to match the first letter of a string, then use one * at the end of your Pattern. If you want to search for your character or string in any position in the String, use two * characters – one at the start of the Pattern and one at the end.

? A placeholder to indicate the location in the String, where your Pattern will start trying to match with a character or string .

[abc]	Match any character specified between the bracket symbols.
[x-y]	Match any character that exists in the interval x-y.
^ [x-y]	Match any character that does not exist in the interval x-y.

Additional Considerations: Any Pattern that uses the placeholder (?) does not need the initial *, because you will specify the exact location(s) in the string for the matching search, when you create the Pattern. The LIKE function is not case sensitive in most situations. However, any Pattern that uses the interval syntax ([x-y]) is case sensitive in relation to its String.

Here is an example:

```
PUBLIC SUB Main()
PRINT "Rittinghouse" LIKE "R*"
END
```

The console responds with:

```
TRUE
```

The special character \ prevents the next character following it in a string from being interpreted as a generic part of the string. Think of the \ character as a control code. Try this:

```
PUBLIC SUB Main()
PRINT "Samson" LIKE "S*"
PRINT "Gambas" LIKE "?[Aa]*"
PRINT "Leonard" LIKE "G[Aa]\\"*
PRINT "Alfred" LIKE "G[Aa]*"
END
```

The console responds with:

TRUE
TRUE
FALSE
FALSE

For a deeper understanding of LIKE, try the following variations of the LIKE statement. Between the PUBLIC SUB MAIN() and END statements, type these PRINT statements :

PRINT "Simple" LIKE "si*"

This returns TRUE. A single "*" character at the END of the search pattern lets you find a character (or string) in the INITIAL POSITION ONLY of the string of the string being searched. It is not case sensitive.

PRINT "Gambas" LIKE "s*"

This returns FALSE. If a character in the string is possibly NOT in the initial position, you need to put one "*" character at EACH END of the search pattern.

PRINT "Gambas" LIKE "*s*"

This returns TRUE because when you have "*" at BOTH ends of the search pattern, you can match the "s" in any position.

PRINT "Simple" LIKE "*impl*"

This returns TRUE because the pattern to be matched can be a string, as well as a single character.

PRINT "Gambas" LIKE "ambas"

This returns FALSE because "ambas" is literally not "Gambas". Without the "*" character, the search pattern will only match the string if it is literally the same as the search pattern.

PRINT "Gambas" LIKE "gamBAS"

This returns TRUE since "Gambas" is literally "gamBAS", so no "*" is needed to return a match. Again, the LIKE function is (usually!) not case sensitive.

PRINT "Gambas" LIKE "*bb*"

This returns FALSE. The search pattern "bb" is not matched -- the string only contains one

"b".

PRINT "Gambas" LIKE "*[bbcyn]*"

This returns TRUE. The [...] (bracket symbols) allow the search pattern to match ANY single character of a string inside the brackets to any of the same character in the string.

PRINT "Gambas" LIKE "??mb*"

This returns TRUE. The "?" (question mark) is a placeholder. It allows you to search for a specific character or substring in a specific position (counting from the left) in the String being searched.

PRINT "Gambas" LIKE "?a???[psy]*"

This returns TRUE. The "?" holds the place of the 1st, 3rd, 4th and 5th positions of the string being searched. "a" is being searched for in 2nd position, and the [...] allows either "p" or "s" or "y" to be matched to the 6th position.

PRINT "Simple" LIKE "??p*"

This returns FALSE. This search pattern looks for "p" in the 3rd position. In the word "Simple", the actual location is in the 4th position.

PRINT "iDealize" LIKE "*[B-D]*"

This returns TRUE. Uppercase "D" lies in the interval B-D.

PRINT "idealize" LIKE "*[b-d]*"

This returns TRUE. Lowercase "d" lies in the interval b-d.

PRINT "iDealize" LIKE "*[b-d]*"

This returns FALSE. Uppercase "D" does not lie in the interval b-d. **Note that only in interval statements (Ix-yJ) is the LIKE function case sensitive!** In order to match letters in mixed uppercase and lowercase strings, you need to test the string twice with LIKE.

PRINT "CARBON" LIKE "*^P-V*"

This returns FALSE -- "R" lies in the interval "P-V".

PRINT "Plum" LIKE "*^s-w*"

This returns FALSE -- "u" lies in the interval "s-w"

```
PRINT "roman" LIKE "??[^n-r]*"
```

This returns TRUE -- "m", the letter in the 3rd position of the string, does NOT lie in the interval "n-r", even though the first two letters of the string DO lie in the interval.

```
PRINT "Leonard*o" LIKE "*\\**"
```

This returns TRUE. This search pattern looks for "*" in the string. (Think of "Leonard*o" as a hip-hop name, because otherwise I can't think of a reason to search for "*".)

Since "*" is used in the LIKE function syntax, we need to distinguish our "*" from the functional symbol "*". To indicate the correct "*", we deactivate it as a functional symbol, by placing a control code "\\" in front of it. But since "*" can also be a special character or code, we use another "\\" to deactivate that. The result is a search pattern like this: "*..*..*", where the * that shown in **bold** is the character that we want to match. *Alternatively, and perhaps preferentially, you can use this notation with equal success:* PRINT "Leonard*o" LIKE "*[*]*"

The **NOT** operator is used to return a result of an expression. The format is as follows:

Result = NOT Expression

When Gambas evaluates this expression, it computes the logical NOT of the expression. If the expression is a string or an object, Gambas returns TRUE if the expression is NULL. Here are some examples to try on your console:

```
PUBLIC SUB Main()
    PRINT NOT TRUE
    PRINT NOT FALSE
    PRINT NOT 11
    PRINT NOT "Gambas"
    PRINT NOT ""
END
```

The console responds with:

```
False
True
-12
False
True
```

String Operators

In Gambas, when you want to compare or concatenate strings, you can use the string operators. These operators allow you to concatenate strings and file paths, perform LIKE operations as explained above, determine if strings are equal or not equal, less than, greater than, less than or equal to, or greater than or equal to each other. Here are some things to try using Gambas console:

```
PUBLIC SUB Main()
    DIM a AS String
    DIM b AS String
    a = "ham"
    b = "burger"
    PRINT "====> "; A ; B
    a = "Gambas"
    b = 1
    PRINT "====> "; A ; " is number " ; B
END
```

The console responds with:

```
====> hamburger
====> Gambas is number 1
```

The following table of string operations is shown for your convenience.

Gambas String Operations

String Operation	Result to determine
String & String	Concatenate two strings.
String &/ String	Concatenate two strings that contain file names. Add a path separator between the two strings if necessary.
String LIKE Pattern	Perform pattern matching using the LIKE operator.
String = String	Determine if the two strings are equal.
String <> String	Determine if the two strings different.
String < String	Determine if the first string is lower than the second string
String > String	Determine if the first string is greater than the second string
String <= String	Determine if first string is lower or equal to second string
String >= String	Determine if first string is greater or equal to second string

Note: all string operator comparisons are case sensitive.

A Beginner's Guide to Gambas – Revised Edition

Note that Gambas is not case-sensitive in the use of your variable names, allowing the variables “a” and “b” to be used as “A” and “B”. To make it easy on yourself, it is better to keep your code as consistent as possible. However you dimension your variables, keep consistent spellings as you write your programs.



Chapter 3 - Keywords and Program Flow Control

In Gambas, the programmer controls the program by using keywords and conditional expressions. Gambas is an event driven language. The programmer has the ability to direct what happens whenever any event occurs. The programmer uses Gambas keywords (which are reserved words with very specific syntax) to create instructions that guide what the program will do in a given circumstance. Conditionals are tests of an expression or a variable. The test could be an evaluation of the result of an operation or comparison of equality, for example. Gambas uses about 250 keywords and native functions. The following table shows a small subset of the Gambas keywords currently supported:

BREAK	CASE	CONTINUE	DEC	DEFAULT	DO
ELSE	END	ENDIF	FOR	FOR EACH	GOTO
IF	INC	LOOP	NEXT	PRINT	QUIT
REPEAT	RETURN	SELECT	STEP	TRUE	THEN
TO	UNTIL	WAIT	WEND	WHILE	WITH

The best way to understand what these keywords and conditionals mean is to gradually introduce them with examples. Rather than go through the list of keywords in alphabetical order, let's take the approach of introducing them based on the type of functionality they support. We will start with the most basic statements and commands and progress through the more complex ones as we continue. Let's start with the Gambas PRINT statement.

The PRINT Statement

The PRINT statement prints expressions to the standard output. The expressions are converted to strings by the Str() function. PRINT takes the format of:

PRINT Expression[(:)] Expression ...] [(:)]

Brackets used in the syntax definition above indicate optional parameters. If there is no semi-colon or comma *after the last expression*, a newline character is automatically printed after the last expression. If a comma is used instead of a semi-colon to separate expressions, then a tab character (ASCII code 9) is printed between output values to separate the expressions. PRINT can also be used to direct output to a file. We will discuss printing to files when we get to the section on Input and Output. When using PRINT to write output to a file, expressions are sent to the stream *File* and this format is used:

PRINT # File, Expression[(:)] Expression ...] [(:)]

A Beginner's Guide to Gambas – Revised Edition

Here is something to try using PRINT:

```
PUBLIC SUB Main()
    DIM b AS Integer
    b = 1
    PRINT "====> " & "b is: " & B
    PRINT "====> ", " b is:", B      'inserts a tab
    PRINT "====> "; "b is: " & B     'catenates without whitespace
END
```

The console displays the following output:

```
====> b is:1
====>   b is:1
====> b is:1
```

Note above that the variable “b” is also written as “B”. For variable names, the case is not important, but it may be confusing to switch between cases.

The IF Statement

The IF statement is used to make a decision. IF is one of the most common structures used by programmers to make comparisons and decisions based on the result of that comparison. Using IF logic, your program can perform comparisons using operators you have learned about and subsequently make decisions based on the result of such comparisons. IF takes the general form of:

```
IF Expression THEN
    do something...
[ ELSE IF Expression THEN
    do something else... ]
[ ELSE
    do something completely different... ]
ENDIF
```

Here is an example using IF that you can try on the console:

```
PUBLIC SUB Main()
    DIM b AS Integer
    b = 1
    IF b = 1 THEN
        PRINT "====> " & "Gambas is number " & B;
    ELSE IF b <> 1 THEN
        PRINT "Should not print me!";
```

```
ELSE
    PRINT "Something bad is happening"
ENDIF
END
```

The console responds with:

```
==> Gambas is number 1
```

You can write a complete **IF ... THEN** statement on one line, provided that there is only one result, and it is written just after the **THEN** keyword. For example:

```
PUBLIC SUB Main()
    DIM a AS Integer
    DIM b AS Integer
    a = 10
    b = 1
    IF b < a THEN PRINT "Yes, 1 is still less than 10."
END
```

The console responds with:

```
Yes, 1 is still less than 10.
```

Furthermore, you can take a shortcut by putting several test expressions on the same line:

```
PUBLIC SUB Main()
    DIM a AS Integer
    DIM b AS Integer
    DIM C AS String
    DIM D AS String
    a = 10
    b = 1
    C = "red"
    D = "yellow"
```

```
IF b < a AND IF C = "red" AND IF D = "yellow" THEN PRINT "Every condition on this line is TRUE, so the IF..THEN statement returned TRUE." 'of course, using this method, ALL of this text must be on one line in the IDE
IF b > a OR IF C = "green" OR IF D = "yellow" THEN PRINT "The IF..THEN statement returned TRUE when it reached the first case of a TRUE condition."
END
```

The console responds with:

**Every condition on this line is TRUE, so the IF..THEN statement returned TRUE.
The IF..THEN statement returned TRUE when it reached the first case of a TRUE condition.**

Note: You are not allowed to mix AND IF and OR IF expressions on the same line.

The SELECT / CASE Statement

The SELECT statement evaluates an expression, comparing it to each CASE specified, and will execute the code enclosed in the matching CASE statement if the expression evaluated is TRUE. If no CASE statement matches the expression under evaluation, the DEFAULT or CASE ELSE statement is executed. The SELECT/CASE statement allows a programmer to build a code block capable of evaluating many expression results without having to code an excessive amount of IF/ELSEIF/ELSE statements. The format of the SELECT statement is:

```
SELECT Expression  
[CASE Expression [, Expression ...]  
[CASE Expression [, Expression ...]  
[CASE ELSE | DEFAULT ...]  
END SELECT
```

Here is some code to show you how to use the SELECT statement:

```
PUBLIC SUB Main()  
  
DIM w AS Integer  
  
w = 1  
' START: is a LABEL, used with GOTO explained below.  
START:  
PRINT "The value of w is: " & w  
  
SELECT CASE w  
CASE 1  
    INC w  
    GOTO START  
CASE 2  
    INC w  
    GOTO START  
CASE 3  
    INC w  
    GOTO START  
CASE ELSE  
    PRINT "The variable w has no handler for: " & w
```

```
END SELECT  
PRINT "Final value of w IS: " & w  
  
END
```

The console should respond with the following output:

```
The value of w is: 1  
The value of w is: 2  
The value of w is: 3  
The value of w is: 4  
The variable w has no handler for: 4  
Final value of w IS: 4
```

Because the variable w was incremented to the value of 4 and there was no **CASE** to handle that value, the **CASE ELSE** block is executed. **CASE ELSE** can also be written as **CASE DEFAULT**. Either way, it is where code defaults to when no case satisfies the value of the variable being checked.

GOTO and LABELS

Note the use of the LABEL named **START**: in the previous code example. The colon must be used with the label name and it must follow it without any spaces between the label and the colon. Labels are the targets where a **GOTO** instruction will direct program flow. While the use of **GOTO** should be judicious it is sometimes necessary. It is used in the above example to demonstrate how to use **GOTO** and **LABELS**. Later, when we study looping structures, we will rewrite this code to use a Gambas looping mechanism.

The FOR / NEXT Statement

Many times when writing code, programmers find the need to iterate through a set of values (called looping) to process data. The **FOR** statement is a looping statement commonly used in programs. It takes the general form of:

```
FOR Variable = Expression TO Expression [ STEP Expression ] ... NEXT
```

The **FOR** statement repeats a loop while at each iteration of the loop incrementing a variable. Note that the variable must be a numeric data-type (i.e., a byte, a short, an integer or a floating point number) and it must be declared as a local variable. If the initial expression the **FOR** statement evaluates is higher than the **TO** expression (for positive **STEP** values) or if the initial expression is less than the **TO** Expression (for negative **STEP** values) the loop will not be executed at all. The **STEP** keyword allows the programmer to define the size of the interval incremented between loop iterations. Here is an example:

```
PUBLIC SUB Main()
DIM I AS Integer
DIM J AS Integer
J=1
FOR I = 1 TO 21 STEP 3
    PRINT "LOOP iteration: " & J & ", I is equal to: " & I
    INC J
NEXT
PRINT "J IS: " & J & " and I is: " & I
END
```

This code results in the following output:

```
LOOP iteration: 1, I is equal to: 1
LOOP iteration: 2, I is equal to: 4
LOOP iteration: 3, I is equal to: 7
LOOP iteration: 4, I is equal to: 10
LOOP iteration: 5, I is equal to: 13
LOOP iteration: 6, I is equal to: 16
LOOP iteration: 7, I is equal to: 19
J IS: 8 and I is: 22
```

Note that once the value of J exceeds the test of 21 in the loop, the loop stops and code flow moves past the **NEXT** statement. The value of J remains unchanged when exiting the loop. Experiment with the code above, changing the STEP value from 3 to 1, for example. Try to modify the **FOR** statement like this:

FOR I = 21 to 1 STEP -1

Gambas also provides a **FOR EACH** looping structure that allows you to iterate through the values of a collection, array, enumerable classes, etc., without the need to maintain an integer counter. We will fully cover the details of **FOR EACH** when we discuss collections later in this book. At this point in your introduction to Gambas, you are encouraged to play around with the code in the examples and change variable values, practice using the **PRINT** statement, etc. Get to know the console and you will quickly learn it is a great tool to test expressions, code segments, etc. Do not be afraid to experiment with Gambas! The very worst thing that can happen is your program blows up and you have to restart your computer – unlikely, but indeed possible. Play around and get comfortable with Gambas. It is the best way to learn.

DO [WHILE] LOOP

The **DO [WHILE] LOOP** structure begins execution of a loop that will not end until a condition is met or code within the loop structure forces an exit. The code executed in the

loop is delimited by the keywords **DO** and **LOOP**. If the optional keyword **WHILE** is not specified, the loop would execute forever (*an infinite loop*) or until some condition within the loop structure forced an exit. If the optional keyword **WHILE** is specified, the loop is stopped once the evaluated result of the expression becomes **FALSE**. In other words, *while* the results of this expression are **TRUE**, continue to iterate (*do*) the loop. If the expression is **FALSE** when the loop is started, the loop will not be executed at all. Here is the format of the DO LOOP:

DO [WHILE Expression]

...

LOOP

The following code example should help you better understand how to use the **DO ... WHILE LOOP**:

```
PUBLIC SUB Main()
DIM a AS Integer
a = 1
DO WHILE a <= 5
IF a = 1 THEN
    PRINT "Hello World, looping " & a & " time."
ELSE
    PRINT "Hello World, looping " & a & " times."
ENDIF
INC a
LOOP
DEC a
PRINT
PRINT "Goodbye World, I looped a total of:" & a & " times."
END
```

The console prints the following:

```
Hello World, looping 1 time.
Hello World, looping 2 times.
Hello World, looping 3 times.
Hello World, looping 4 times.
Hello World, looping 5 times.
```

```
Goodbye World, I looped a total of: 5 times.
```

WHILE [Expression] **WEND** Loops

This loop structure begins a loop delimited by the **WHILE** ... **WEND** instructions. The loop is repeated while *Expression* is TRUE. If, on initial entry, the expression is FALSE, the loop is never executed. The **DO WHILE LOOP** and **WHILE** ... **WEND** structures are equivalent. Try this code:

```
PUBLIC SUB Main()
    DIM a AS Integer
    a = 1
    WHILE a <= 5
        IF a = 1 THEN
            PRINT "Hello World, WHILE...WEND looping " & a & " time."
        ELSE
            PRINT "Hello World, WHILE...WEND looping " & a & " times."
        ENDIF
        INC a
    WEND
    DEC a
    PRINT "Goodbye, WHILE...WEND looped a total of: " & a & " times."
END
```

The console should print the following:

```
Hello World, WHILE...WEND looping 1 time.
Hello World, WHILE...WEND looping 2 times.
Hello World, WHILE...WEND looping 3 times.
Hello World, WHILE...WEND looping 4 times.
Hello World, WHILE...WEND looping 5 times.
Goodbye, WHILE...WEND looped a total of: 5 times.
```

The **REPEAT UNTIL** loop

This loop structure begins to execute code that is delimited by the keywords **REPEAT** and **UNTIL**. A repeat loop will always execute at least once, even if the **UNTIL** value is initially FALSE. Here is a sample to try on the console:

```
PUBLIC SUB Main()
    DIM a AS Integer
    a = 1
    REPEAT
        IF a = 1 THEN
            PRINT "Hello World, REPEAT UNTIL looping " & a & " time."
```

```
ELSE
    PRINT "Hello World, REPEAT UNTIL looping " & a & " times."
ENDIF
INC a
UNTIL a > 5
DEC a
PRINT
PRINT "Goodbye, REPEAT UNTIL looped a total of: " & a & " times."
END
```

The console should print the following:

```
Hello World, REPEAT UNTIL looping 1 time.
Hello World, REPEAT UNTIL looping 2 times.
Hello World, REPEAT UNTIL looping 3 times.
Hello World, REPEAT UNTIL looping 4 times.
Hello World, REPEAT UNTIL looping 5 times.

Goodbye, REPEAT UNTIL looped a total of: 5 times.
```

Defining and Using Arrays in Gambas

There are two kinds of arrays that can be used in Gambas. The first type of Gambas array works like arrays used in the Java programming language. The other type of array that is used in Gambas is known as a Native array. When using *Java-like* arrays, you must remember that the arrays are objects of the following classes: Integer[], String[], Object[], Date[], and Variant[]. All of these types of *Java-like* arrays can have only one dimension. You declare them like this:

```
DIM MyArray AS NEW Integer[]
```

Arrays are always initialized as void at startup. They are dynamic, and have a lot of useful methods applying to them. When using single-dimension arrays, these are the right choice. *Native arrays* on the other hand can support multidimensional implementations or arrays with arrays objects of the following classes: Integer, String, Object, Date, and Variant. They are declared like this:

```
DIM MyArray[Dim1, Dim2, ... ] AS Integer
DIM MyArray[Dim1, Dim2, ... ] AS String
DIM MyArray[Dim1, Dim2, ... ] AS Variant
```

Each native datatype has an associated array data-type whose name is the name of that native datatype followed by square brackets: **Boolean[]**, **Byte[]**, **Short[]**, **Integer[]**,

Single[], Float[], String[], Date[], Variant[], Pointer[], and **Object[]**. Native arrays can have up to eight dimensions. They are NOT objects. They are allocated on the stack if you declare them local to a function. They are allocated inside the object data if you declare them as global in scope. Native arrays are not dynamic. They can't grow or shrink once they are declared. You can only set or get data from an element in these types of arrays. Here is an example of using a three-dimensional native array in Gambas. In this example, the array is filled with integer values ranging from 0 to 26. Enter this code in your console code window:

```
PUBLIC SUB Main()
DIM i AS Integer
DIM ii AS Integer
DIM iii AS Integer
DIM narMatrix[3, 3, 3] AS Integer
FOR i = 0 TO 2
    FOR ii = 0 TO 2
        FOR iii = 0 TO 2
            PRINT i, ii, iii & " ==> "; 'note that ";" prevents a newline.
            narMatrix[i, ii, iii] = i*9 + ii*3 + iii
            PRINT narMatrix[i, ii, iii]
        NEXT
    NEXT
NEXT
END
```

When you execute this code, your output in the console window should be similar to what is shown here:

```
0 0 0 ==> 0
0 0 1 ==> 1
...
2 2 1 ==> 25
2 2 2 ==> 26
```

To make the **FOR..NEXT** logic very clear, here is that code block with vertical lines connecting the paired keywords of each separate loop:

```

FOR i = 0 TO 2
    FOR ii = 0 TO 2
        FOR iii = 0 TO 2
            PRINT i, ii , iii & " ==> ";
            1 2 3 narMatrix[i, ii, iii] = i*9 + ii*3 + iii
            PRINT narMatrix[i, ii, iii]
        NEXT
    NEXT
NEXT

```

Figure 18: Clarifying the For Next loop structure.

From loop iteration 0 to 2, there are three execution steps. Each of the three **FOR..NEXT** loop iterations executes three cycles, for each of the three times that control passes to it. Therefore, $3 \times 3 \times 3$ cycles equals 27 cycles. Starting the loop counter with $i = 0$ and iterating from 0 to 26 iterations makes a total of 27 cycles.

Collections

Collections are groups of objects implemented with a hash table. Collection objects use keys that are implemented as string data-types. The data values corresponding to any collection key is a Variant data-type. Objects in a collection are enumerable. **NULL** is used when nothing is associated with a given key. Consequently, associating **NULL** with a key has the same effect as removing it from the collection. The size of the internal hash table grows dynamically as data is inserted. This class is created using this general format:

```

DIM hCollection AS Collection
hCollection = NEW Collection ( [Mode AS Integer] )

```

Note the word **EACH** after the **FOR** statement. The **FOR EACH** construct is discussed next. Also, we will go into much greater detail about collections later in this book when we talk about object-oriented programming. We will develop an application that makes extensive use of collections. For now, just note that collection elements are enumerated *in the order* in which they were inserted into the collection. However, if you replace the value of an already inserted key, the original insertion order is kept.

The FOR EACH Statement

The **FOR EACH** statement executes a loop while simultaneously enumerating an object. The *Expression* must be a reference to an enumerable object such as a collection or an array. The order of enumeration is not necessarily predictable. The general format of the statement is:

```

FOR EACH Expression ... NEXT
FOR EACH Variable IN Expression ... NEXT

```

This syntax must be used when *Expression* is an enumerable object that is not a container. For example, an object returned as the result of a database query. Furthermore, when we say that the object is “enumerable”, we mean that it has a built-in list structure, even if the numerical order of the list items is only created when the FOR EACH loop executes. In the collection below, we can use integers as keys, but they are automatically converted to the string data-type when we assign them as keys. The statements above create a new collection object. Enter the following code into your console code window and run it.

```
PUBLIC SUB Main()
DIM MyDict AS NEW Collection
DIM strElement AS String

MyDict["absolute"] = 3
MyDict["basic"] = 1
MyDict["carpet"] = "2"
MyDict["raindrop"] = "cat"

FOR EACH strElement IN MyDict
    PRINT strElement & " ";
NEXT
END
```

Your console should display this output:

```
3 1 2 cat
```

At this point, you should be getting a pretty good feel for the syntax and structure of Gambas. We are going to move away from console-based (terminal) applications and return to our first project to begin exploring the Gambas ToolBox and learn how to develop a GUI-based program. It is time to take a break and when you come back to the next chapter, you will be refreshed and ready to code. Ah, heck! If you just can't wait to jump in, go ahead and turn the page!



Chapter 4 – The Gambas ToolBox

The Gambas ToolBox consists of many controls. As of this writing using release (2.22), the following controls are currently supported:

Select	RadioButton	TextArea	TableView	HPanel	Expander
Label	ToggleButton	ListView	ValueBox	VPanel	ListContainer
TextLabel	Toolbutton	TreeView	ColorChooser	HSplit	SidePanel
Separator	Slider	IconView	DateChooser	VSplit	ToolPanel
PictureBox	ScrollBar	ColumnView	DirChooser	Panel	Wizard
MovieBox	ListBox	GridView	FileChooser	Frame	Embedder
ProgressBar	ComboBox	ColorButton	FontChooser	TabStrip	TrayIcon
Button	TextBox	DirView	HBox	ScrollView	Timer
CheckBox	SpinBox	FileView	VBox	DrawingArea	

As you can see, there is a wide selection of tools for you to play with in Gambas. Our introduction to the tools (*which are also called controls*) will teach you about what each control does, what properties, methods, and events you can use to direct the behavior of the interface you are building, and how to code the control to make it work exactly as you desire.

Along the way, we will be building several simple projects to get you more experienced in using the Gambas IDE. All of the controls in the ToolBox are built-in using the **gb.qt component**. This component implements the Graphical User Interface classes. It is based on the Qt library. Here is a list of some of the various functionality the **gb.qt component** provides:

Clipboard	IconView
Containers	Keyboard and Mouse
Drag and Drop	Menus
Drawing	ListView, TreeView and ColumnView
Fonts	Printing

We will cover other components later in this book. In Figure 8, you see an example of a Form. This is the default toolkit for the gb.qt component. The toolkit for GTK+ is somewhat different. When you start a Gambas project and select a “Graphical application”, Gambas will load a default toolkit based on your desktop system: Qt for the KDE desktop, GTK+ for Gnome. Therefore, if you prefer a specific toolkit other than the default for your system, you must choose it at project creation.

NOTE: *For several reasons, Qt is the more reliable and complete toolkit for Gambas at the time of this writing. For now, let's take a quick look at the Qt ToolBox again and review the icons therein.*

A Beginner's Guide to Gambas – Revised Edition



Figure 19: The Gambas Toolbox.

From the top left of Figure 1, the controls in the ToolBox (Row 1) are:

- ✓ Select
- ✓ Label
- ✓ TextLabel
- ✓ Separator
- ✓ PictureBox
- ✓ MovieBox
- ✓ ProgressBar
- ✓ Button
- ✓ CheckBox
- ✓ RadioButton
- ✓ ToggleButton
- ✓ ToolButton
- ✓ Slider
- ✓ ScrollBar

On the second row you will find:

- ✓ ListBox
- ✓ ComboBox
- ✓ TextBox
- ✓ SpinBox
- ✓ TextArea
- ✓ ListView
- ✓ TreeView
- ✓ IconView
- ✓ ColumnView
- ✓ GridView
- ✓ ColorButton
- ✓ DirView
- ✓ FileView
- ✓ TableView

On the third row we have:

- ✓ ValueBox
- ✓ LCDNumber
- ✓ Dial
- ✓ Editor
- ✓ TextEdit

The general approach to programming Gambas is to design the layout of your forms, place the controls that make up your interface to the user on the form, determine and handle the various events that can occur with each control using a combination of built-in Gambas methods and your own code, and direct the operations of the program using input/output data. If it sounds simple, it is because once you understand the mechanics of it all, it is simple. Once you get used to programming event-driven applications in Gambas, it will become second nature to you.

The first group of controls we are going to learn about are text-oriented in nature and serve to either display or gather text data or to generate an *event* that you can write code for, such as pushing a button. These controls are:

- ✓ Button
- ✓ Label
- ✓ TextLabel
- ✓ TextBox
- ✓ InputBox
- ✓ TextArea

The Button Control

This class object inherits its attributes from the **Control** class, as do all the controls in the Gambas ToolBox. Using this control implements a push button on a form. A push button can display text, a picture, or both. You have the ability to set one button inside a window to be the **Default button**. Then, when the user presses the **RETURN** key, it will activate that button automatically. Also, you can set one button inside a window to be the **Cancel button**. Pressing the **ESC** key will activate the Cancel button automatically. This class is creatable, meaning that you can write code to dynamically generate a button on a form at run-time. To declare a button object, the format below is used:

```
DIM hButton AS Button  
hButton = NEW Button ( Parent AS Container )
```

The Container, in most cases, will be the form where you place the button. It could be any container object, however. This code will create a new push button control. Note the variable name is preceded by the lowercase letter *h*. This is a standard convention

programmers use to refer to the handle (*handles are really references to something*) of an object. Since Gambas supports object-oriented programming, we will gradually introduce OO (*Object-Oriented*) concepts to you as we encounter them. The term "**object**" in Gambas is a short way of referring to a data structure that provides properties, variables, methods and events. One kind of object is a "form", which is an object of the built-in Gambas class named "Form". A form might contain other objects like buttons or checkboxes. Objects aren't necessarily graphical in nature -- another kind of object might be an array of integers or strings.

A "**class**" in Gambas is the description of all of the public properties, methods, and events belonging to objects. Of course, there can be many objects collected in a class, including forms. A class can contain a form's description, including the buttons, boxes, etc. that the user interacts with, and all of the events that are connected with those objects.

Gambas' built-in classes are not directly visible, but the classes that a programmer creates in the IDE are contained in class files. These files are in plain text format, visible to the programmer either in the IDE or separately with a text editor. In addition, forms and modules are classes created by the programmer – thus, form files and module files are also stored in plain text, and are also directly readable with a text editor. These files are all located in the folders for each Gambas programming project. Therefore, when we refer to the programmer writing code for a class, we are referring both to the programming concept of a "class" as well as an actual human-readable class file. OO will be covered in much greater detail in Chapter 11 of this book.

Generally, objects are created from a class by assigning a copy of the object to a variable, in this case **hButton**. This process, known as **instantiating** an object occurs when hButton is assigned a value using the NEW keyword. What the line below means is hButton will be instantiated as a NEW Button object and **Parent** is the property (*read only*) that will read a method (think of methods as functions that set or get a value) that is implemented inside the Container class. The Container class is used because it is the parent class of every control that can contain other controls. Now, this line of code should make perfect sense to you:

hButton = NEW Button (Parent AS Container)

Common Control Properties

Button properties are attributes that you can set or read using the property window or by using code in your program. For example, the line of code below will set the Text property of a button:

hButton.Text = "OK"

Note the format of the statement above. **Control.Property = Expression** is the standard convention used for setting or getting attribute information from a control. In the term

Control.Property you see an example of “dot notation”. This is used in Object-Oriented Programming, where the connection between the object and its properties or methods is indicated by a “dot”. Many of the properties for controls are common among all controls, so we will take the time now to explain all of the button properties here. Later in the book, we will only explain the properties that are unique to a given control when we encounter them.

Background is defined as **PROPERTY Background AS Integer**

This integer value represents the color used for the control background. It is synonymous with the Background property., which will be eliminated in Gambas3. Note that **PROPERTY** is a predefined data-type used internally in Gambas. You can use the Gambas predefined constants for color to set common color values:

Black	Blue	Cyan	DarkBlue	DarkCyan	DarkGray
DarkGreen	DarkMagenta	DarkRed	DarkYellow	Default	Gray
Green	LightGray	Magenta	Orange	Pink	Red
Transparent	Violet	White	Yellow		

To set the Background property for hButton to red, you would use this code:

hButton.Background = Color.Red

Alternatively, if you know the RGB (red, green, blue) or HSV (hue, saturation, value) values for a specific color, Gambas provides a means to convert those values to an integer value that can be passed on to the Background (*or other color-related*) property. The class Color provides two methods, RGB and HSV that you can use. Here is how those functions are defined:

STATIC FUNCTION RGB (R AS Integer, G AS Integer, B AS Integer [, Alpha AS Integer]) AS Integer

The **RGB** function returns a color value from its red, green and blue components. For the HSV function, use this:

STATIC FUNCTION HSV (Hue AS Integer, Sat AS Integer, Val AS Integer) AS Integer

HSV returns a color value from its hue, saturation and value components. To use one of these functions to set the button's background color, here is what you could code:

hButton.Background = Color.RGB(255,255,255)

This would set the button's background color to white. This is most useful when you are trying to set colors whose values fall outside the default color constant values provided with Gambas.

Border - This property is another common attribute found on many controls. This class is static. The constants used by the Border property include the following: *Etched*, *None*, *Plain*, *Raised*, and *Sunken*. To set the border property for our button, we could use this code:

hButton.Border = Border.Plain

Cancel is defined as **PROPERTY Cancel AS Boolean** and it is used to stipulate whether or not the button is activated when the ESC key is pressed. The following code would trigger this button if the user pressed the ESC key on their keyboard.

hButton.Cancel = TRUE

If the Cancel property is used in your program, it would allow you to handle the event and gracefully exit the form just as if the user had clicked a Quit, Exit or Cancel button.

Caption is defined as **PROPERTY Caption AS String** and the Caption property is synonymous with the Text property. The following code

hButton.Caption = "OK"

is the same as

hButton.Text = "OK"

and can be used interchangeably. Let's try some code now by stating Gambas. Next, open up the project we created named FirstProject. From the TreeView, double-click your mouse cursor on the FMain form and the blank form will open up. Double-click anywhere on the blank form and you will see the code window. It should look like this:

```
' Gambas class file
PUBLIC SUB Form_Open()
```

END

Now, enter this code between the **PUBLIC SUB Form_Open()** and **END** statements so it looks like this:

```
PUBLIC SUB Form_Open()
DIM hButton AS Button
hButton = NEW Button (ME) AS "hButton"
hButton.X = 215
hButton.Y = 60
hButton.Width = 200
```

```
hButton.Height = 40
hButton.Enabled = TRUE
hButton.Text = "Click or press ESC to Quit"
hButton.Border = TRUE
hButton.Default = TRUE
hButton.Cancel = TRUE
hButton.Show
END
```

After the PUBLIC SUB Form_Open()'s END statement, add this routine:

```
PUBLIC SUB hButton_Click()
    FMain.Close
END
```

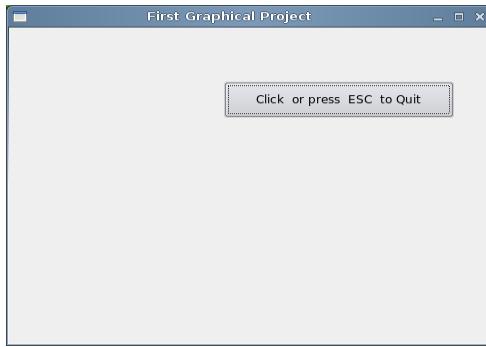


Figure 20: First button code results.

Once you have entered the code above, click the green program run button and you should see something similar to what is shown in the figure above. Simply click the button or press the ESC key to exit this program. Congratulations! You have just created your first GUI-based program in Gambas! Yes, it is a little bit ugly but remember, you did it the hard way, using code instead of taking advantage of the Gambas ToolBox. Now, save your project and exit Gambas for now and we will proceed with learning more about the ToolBox.

Programming Tip:

Always try use predefined constants if they are available instead of hard-coding numeric values to ensure portability. For example, key codes and mouse styles have the same names as they are defined as constants, but their numeric values are totally different. The gb.gtk component will not work with gb.qt code if you do not use those predefined constants. Therefore, when writing code, it is imperative to use predefined constants instead of numeric values. If not, the code will not be portable.

Cursor is defined as **PROPERTY Cursor AS Cursor** and you can use the Cursor

property for assigning a custom cursor to a control. This class implements a custom mouse cursor. This class is creatable. Here is how to declare a cursor object variable and instantiate a cursor object:

```
DIM hCursor AS Cursor  
hCursor = NEW Cursor(MyPic AS Picture [, X AS Integer, Y AS Integer])
```

The code format above can be used to create a new cursor from a Picture object and set an optional hot spot. If it is not specified, the default hotspot is the top left of the picture. Cursor has two properties, X and Y, that can be used to set or get the current mouse position. Here is an example that shows you how to use a Cursor object in your program:

```
DIM hPict AS Picture  
DIM hCursor AS Cursor  
hPict = Picture["torric.png"]  
hCursor = NEW Cursor(hPict)  
ME.Cursor = hCursor
```

Default is defined as **PROPERTY Default AS Boolean** and you use the Default property to indicate whether or not the button is activated whenever the RETURN key is pressed. Using the code below, it would force the user to actually click a button rather than hit the RETURN key to activate it. Generally, when creating an OK/Cancel type dialog, you would want to set one of the buttons to a default action using this property. Think about just hitting the RETURN key in a wizard to accept the default settings and move on.

```
hButton.Default = FALSE
```

Design is defined as **PROPERTY Design AS Boolean** and it indicates that the control is in design mode. This means that when the Design property is set to TRUE, controls in design mode just draw themselves and will not react to any input event. This property is used by the Gambas Development Environment to display controls in the form editor. What this really means is you do not need to worry about this property.

Drop is defined as **PROPERTY Drop AS Boolean** and is used to check to see if a control accept the drop from a drag and drop operation. The Drop property is either TRUE or FALSE and you can check the value before trying to drop something onto the control by using code like this:

```
IF hButton.Drop = FALSE THEN  
    hButton.ToolTip = "No Drops allowed"  
ELSE  
    hButton.ToolTip = "Drop something here"  
ENDIF
```

Enabled is defined as **PROPERTY Enabled AS Boolean** and it indicates that the

control is enabled. To disable a control, simply set its Enabled property to FALSE.

```
IF Expression = FALSE THEN  
    hButton.Enabled = TRUE  
ELSE  
    hButton.Enabled = FALSE  
ENDIF
```

The **Expand** property is defined as **PROPERTY Expand AS Boolean** and it indicates that the control is able to expand if it is included in a container that dynamically re-arranges its contents.

Font is defined as **PROPERTY Font AS Font** and it returns or sets the font used to draw text in the control. To set Font property attributes, code like this can be used:

```
hButton.Font.Name = "Lucida"  
hButton.Font.Bold = TRUE  
hButton.Font.Italic = FALSE  
hButton.Font.Size = "10"  
hButton.Font.StrikeOut = FALSE  
hButton.Font.Underline = FALSE
```

The output of the above code would be as follows:



Figure 21: Demonstrating font capabilities.

Foreground is defined as **PROPERTY Foreground AS Integer** and this integer value represents the color used for the control foreground. It is synonymous with the Foreground property. You can use the Gambas predefined constants for color to set color value:

Black	Blue	Cyan	DarkBlue	DarkCyan	DarkGray
DarkGreen	DarkMagenta	DarkRed	DarkYellow	Default	Gray
Green	LightGray	Magenta	Orange	Pink	Red
Transparent	Violet	White	Yellow		

To set the Foreground property for hButton to red, you would use this code:

```
hButton.Foreground = Color.Red
```

As with the Background property, if you know the RGB or HSV values for a specific color, you can pass them to the Foreground property with this code:

```
hButton.Foreground = Color.RGB(255,255,255)
```

This would set the button's foreground color to white.

The **X** property is defined as **PROPERTY X AS Integer** and it is used to set or return the x position of the control. **X** is the same as Left property. The **Y** property is defined as **PROPERTY Y AS Integer** and it is used to set or return the y position of the control (top left corner).

The **Height** property is defined as **PROPERTY Height AS Integer** and it is used to set or return the height of the control. It is synonymous with the **H** property and can be used interchangeably. This value describes how tall the control is from the x position moving down **Height** pixels. The **Width** property is defined as **PROPERTY Width AS Integer** and it is used to set or return the width of the control. It is synonymous with the **W** property and can be used interchangeably. This value describes how wide the control is from the x position to the right **Width** pixels.

The **Handle** property is defined as **PROPERTY READ Handle AS Integer** and is used to return the internal X11 window handle of the control. It is a read-only attribute. It is synonymous with the **Id (PROPERTY READ Id AS Integer)**.

The **Mouse** property is defined as **PROPERTY Mouse AS Integer** and is used to set or return the appearance of the cursor to a predefined image when it hovers inside the boundaries of the control. Predefined values for the Mouse property are shown in the table below:

Cursor shape	Value	Cursor shape	Value	Cursor shape	Value
Arrow	0	SizeH	6	SizeV	5
Blank	10	SizeN	5	SizeW	6
Cross	2	SizeNE	7	SplitH	12
Custom	-2	SizeNESW	7	SplitV	11
Default	-1	SizeNW	8	Text	4
Horizontal	0	SizeNWSE	8	Vertical	1
Pointing	13	SizeS	5	Wait	3
SizeAll	9	SizeSE	8		
SizeE	6	SizeSW	7		

The **Next** property is defined as **PROPERTY READ Next AS Control** and it is used to return the next control having the same parent within a Container. It is useful when

iterating through a series of controls to set focus. We will discuss this property in more detail when we discuss the SetFocus() method. To get the previous control, one would use the **Previous** property, defined as **PROPERTY READ Previous AS Control**. This property returns the previous control having the same parent.

The **Parent** property is defined as **PROPERTY READ Parent AS Control** and is used to return the container that holds the control. It is a read-only property.

The **Picture** property is defined as **PROPERTY Picture AS Picture** and returns or sets the picture shown on a control (i.e., a button). We will discuss the Picture class later in this chapter.

The **ScreenX** property is defined as **PROPERTY READ ScreenX AS Integer** and it returns the position of the left border of the control in screen coordinates. Usually, it is used in conjunction with ScreenY.

The **ScreenY** property is defined as **PROPERTY READ ScreenX AS Integer** and returns the position of the top border of the control in screen coordinates.

The **Tag** property is defined as **PROPERTY Tag AS Variant** and is used to return or sets the control tag. This property is intended for use by the programmer and **is never** used by the component. It can contain any Variant value.

The **Text** property is defined as **PROPERTY Text AS String** and returns or sets the text displayed in the button. It is almost always used and its value can be changed dynamically. We have already used this property in the example above:

```
hButton.Text = "Click or press ESC to Quit"
```

The **ToolTip** property is defined as **PROPERTY ToolTip AS String** and returns or sets the tooltip (*a tooltip is the little box of text that pops up over a control when the mouse hovers over it for a few seconds*).

```
hButton.ToolTip = "Click me!"
```

The **Top** property is defined as **PROPERTY Top AS Integer** and returns or sets the position of the top border of the control relative to its parent.

The **Value** property is defined as **PROPERTY Value AS Boolean** and it is used to activate the control (i.e., make the button click). This is done if you set this property to TRUE. Reading this property will always return a value of FALSE.

The **Visible** property is defined as **PROPERTY Visible AS Boolean** and it indicates if the control is visible or not. Setting this property to FALSE will make it disappear from view. Conversely, setting it to TRUE makes it appear.

The **Window** property is defined as **PROPERTY READ Window AS Window** and returns the top-level window that contains the control.

At this point, we have covered all of the properties for the Button Control and you should have a pretty good understanding of what they are used for and how to use them. Remember that most of these properties are common to many controls. Next, we will cover the methods (things you can do with the button) and provide you some examples of how to use them.

Button Methods

Methods are the built-in routines provided for a control that allow you to do things like show it, hide it, move it, etc. Here are the methods supported by the Button control: **Delete**, **Drag**, **Grab**, **Hide**, **Lower**, **Move**, **Raise**, **Refresh**, **Resize**, **SetFocus**, and **Show**. Usually, a method is called in response to some event. We will discuss events in the next section. Let's take a look at the methods used with Button:

The **Delete** method is defined as **SUB Delete()** and when invoked it destroys the control. It is important to know that a destroyed control becomes an invalid object. What this means is that once you have destroyed it, ensure no other code subsequently references it.

The **Drag** method is defined as **SUB Drag (Data AS Variant [, Format AS String])** and when invoked it starts a drag & drop process. Data is the data to be dragged. It can be a String or an Image. If Data is a text string, then you can specify in Format the MIME TYPE of the text being dragged. For example, MIME TYPE “text/html”.

The **Grab** method is defined as **FUNCTION Grab () AS Picture** and it grabs a picture of the control and returns it.

The **Hide** method and the **Show** method are defined as **SUB Hide()** and **SUB Show()** and they simply hide or show the control. Here is how they would be used:

```
IF Expression = TRUE THEN
    hButton.Hide
ELSE
    hButton.Show
ENDIF
```

The **Lower** method and **Raise** method are defined as **SUB Lower()** and **SUB Raise()**. Lower sends the control to the background of its parent. Raise brings the control to the foreground of its parent. As an example of how these methods work, let's go back to our project, FirstProject. Change the code in the **hButton_Click()** routine. First, select and copy all of the the code in your **PUBLIC SUB Form_Open()..END** block, and paste it into your **PUBLIC SUB hButton_Click()..END** block.

After the hButton.Show line, and before FMain.CLOSE, add the following code:

```
hButton.Text = "Duch!"  
WAIT 2.0  
hButton.Hide  
WAIT 1.5  
hButton.Show  
hButton.Text = "Better now! -- goodbye!"  
Wait 1.0
```

Once you have entered the code, run the program and see what happens. When you click the button it will disappear and the hButton.Text will change. After two seconds, it will reappear and change the label again. Also, note that we slipped in a new command for you to try – **WAIT**. WAIT takes a floating point value as a parameter that represents seconds or fractions of seconds expressed as decimal values. Numbers less than 1 are processed as milliseconds. WAIT 1.5 would pause execution for 1,500 milliseconds. If you specify WAIT without a parameter, it will refresh the interface and it does not allow the user to interact with the application.

The **Move** method is defined as **SUB Move (X AS Integer, Y AS Integer [, Width AS Integer, Height AS Integer])** and is used to move and/or resize the control.

```
hButton.Move(hButton.X - 50, hButton.Y - 20)
```

The **Refresh** method is defined as **SUB Refresh ([X AS Integer, Y AS Integer, Width AS Integer, Height AS Integer])** and is used to redraw the control, or in some instances, just a portion of it.

The **Resize** method is defined as **SUB Resize (Width AS Integer, Height AS Integer)** and is used to resize the control.

The **SetFocus** method is defined as **SUB SetFocus ()** and you call this method to give focus to the control. Focus is roughly equated to where the text cursor would be located. When a control has focus it is usually indicated visually in some manner. For example, a button with focus may have a small dotted line around it to indicate where focus control is on the form:

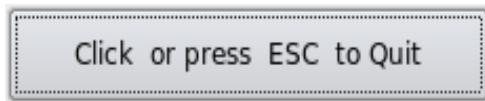


Figure 22: The dotted line indicates focus for a control.

A Beginner's Guide to Gambas – Revised Edition

We have covered all of the methods available for the Button class (control) and will now begin discussing the events for that class. As a change of pace, let's try to learn about events with a more hands on approach. We can do this by building a new program and learning what the events do and how to use them. Click on File>New on the top menu bar, or just click on the blank document icon to start a new project. This will close the current project and initiate the New Project Wizard. Select “Qt graphical application” and go through the wizard just like we did with the project FirstProject. Name this project SecondProjectQT. Once the IDE appears, double-click on the “FMain” icon in the TreeView window. This first form will always be named “FMain.form”. For this project, we are going to place three Label controls on the form, as shown below. We will use the form tools provided in the toolbox in the lower right-hand corner of the IDE.

Create the labels by clicking on the “Label” icon in the toolbox, then clicking and dragging the mouse cursor in the form, to make rectangles. When a form has been created, it will automatically be selected, and you can use the border handles to resize them as appropriate. You can click on these objects at any time and resize or grab-and-move them, and you can also move them by pressing down on your keyboard's cursor (arrow) keys, as well.

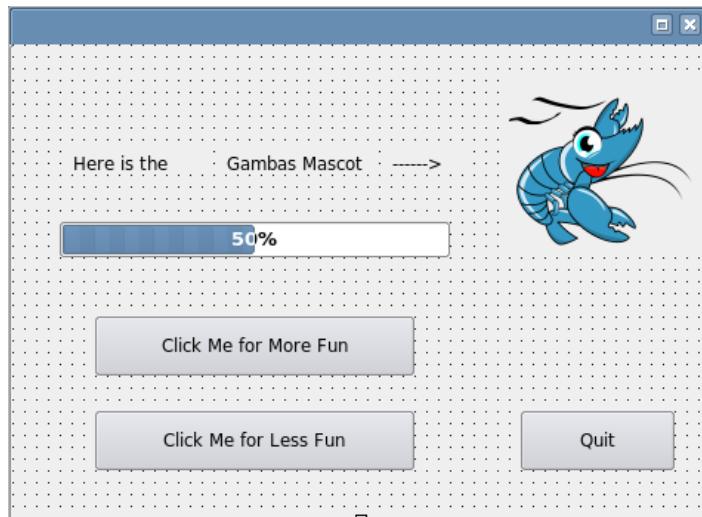


Figure 23: Layout for the FMain form for the second project.

The labels, buttons, etc., will automatically be named in the order that you create them. You can also change the names, if desired. The first label's text caption is set with the Label.Text property – click on the Label1 box which you created, and find “Text” on the Properties tab to the right of the form (if it is not visible, press **F4** on your keyboard, to toggle the tabs into view). The setting for Label1 should be “Here is the”; for Label2 “Gambas Mascot”; and for Label3 “---->”, which points to a PictureBox control (where the shrimp lives). Next, create the PictureBox control **PictureBox1**.

To obtain the graphic of the Gambas mascot, you can go to the official Gambas web site¹² and right click on the mascot. Choose to save the image and save it as “gambas mascot.png” on your computer. Move the file to the data directory in the SecondProject folder. (*A Gambas quirk is that the file explorer dialogs don't seem to be dynamic. Once they have initialized, changes made outside the Gambas environment are not updated dynamically.*) Once you have done this, you can set the Picture property for PictureBox1 to the file named “gambas mascot.png”. At this point, the top of your form should look like the figure below.

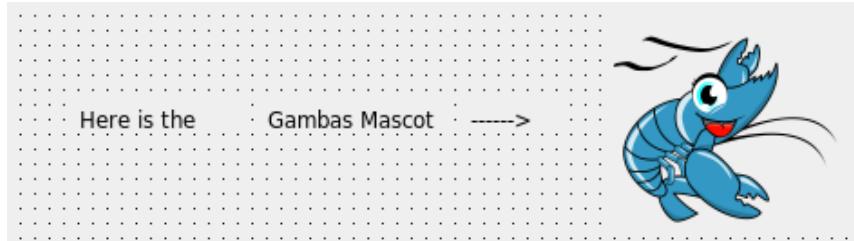


Figure 24: A partially constructed form with our first four controls.

Now, select the **ProgressBar** control from the ToolBox and place it on the form so it looks like that shown in Figure 17 above. Finally, we have three buttons to add to the form. Name these buttons FunBtn, NoFunBtn, and QuitBtn. All of the necessary controls for our example have been added to the form at this point. Try to get the layout to look as close to the picture shown in the figure above as possible. At this point, you may be asking what all of this is going to accomplish? Remember, we are going to learn how to use events with these controls. We have placed the controls where we want them and now the next step is to decide what events we are going to handle.

For this exercise, we are going to handle mouse events when the mouse moves over labels. If the mouse moves off of the main window (**FMain.form**) or back onto it we will handle those events. We will also handle button clicks and the ESC key if it is pressed.

Let's begin by setting the **ProgressBar1.Value** to 50% when the program starts. That is handled with the initialization routine when opening the form at program runtime, as shown below:

```
' Gambas class file
PUBLIC SUB Form_Open()
    ProgressBar1.Value = 0.50
    ProgressBar1.Background = Black
    ProgressBar1.Foreground = Yellow
END
```

¹² The official Gambas site is <http://gambas.sourceforge.net/index.html>.

A Beginner's Guide to Gambas – Revised Edition

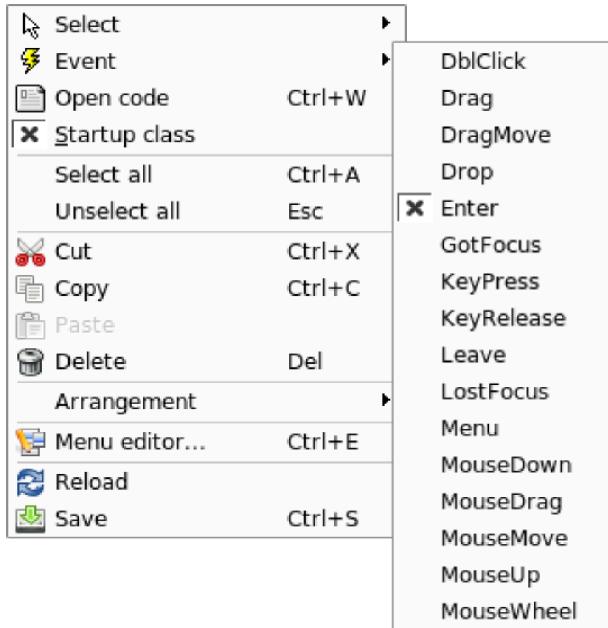


Figure 25: The Gambas Event Menu.

When the mouse moves over the startup form (*our Gambas program on the desktop*), we want to set the three top labels' text properties to blank values so it appears that the text disappears. This requires choosing an *event handler*. To choose an event from the Gambas IDE, move the mouse over the form (but not directly over a control) and click it once. Next, right click the mouse and you should see a menu pop-up, similar to the one shown in the following figure. Choose the Event selection item from the popup and pick the Enter event.

Add this code, as shown below:

```
PUBLIC SUB Form_Enter()
    Label1.Text = ""
    Label2.Text = ""
    Label3.Text = ""
END
```

Repeat this process to choose another event, the **Leave** event, so that when the mouse moves off the program window we can detect that and do something about it. Add this code:

```
PUBLIC SUB Form_Leave()
    ProgressBar1.Value = 0.50
    Label1.Text = "Wasn't that"
    Label2.Text = "some real"
    Label3.Text = "fun stuff?"
END
```

A Beginner's Guide to Gambas – Revised Edition

The next thing we want to do is handle the situation (event) that occurs when the mouse moves over any of our labels. In this event, we are going to change the Label.Text property of the affected label to indicate that a mouse has been detected over the control. Single-click your mouse cursor on the first label and when the handles appear, right click and choose the Enter event when it appears on the Event submenu. In the code window, find the **Label1_Enter()** subroutine and enter this code:

```
PUBLIC SUB Label1_Enter()
    Label1.Text = "Here is the"
END
```

Repeat the same process for the Label2.Text and Label3.Text properties. Your code should look like this:

```
PUBLIC SUB Label2_Enter()
    Label2.Text = "Gambas Mascot"
END
```

```
PUBLIC SUB Label3_Enter()
    Label3.Text = "----->"
END
```

We also want to detect and respond to events when the mouse moves over our mascot picture as well. Click once on the PictureBox1 control and choose the Enter Event. Go to the code window and make the code look like this:

```
PUBLIC SUB PictureBox1_Enter()
    Label1.Text = ""
    Label2.Text = ""
    Label3.Text = ""
    PictureBox1.Border = Border.Plan
END
```

Repeat the process, choosing the **Leave** event and ensure your code is typed in the code editor as follows:

```
PUBLIC SUB PictureBox1_Leave()
    Label1.Text = "Wasn't that"
    Label2.Text = "some real"
    Label3.Text = "fun stuff?"
    PictureBox1.Border = Border.Sunken
END
```

The next control that we want to take care of is the Quit button. Double-click on it and

the code window displays the **QuitBtn_Click()** subroutine, which responds to a Click event. Add the **Form1.Close** line, as shown below.

```
PUBLIC SUB QuitBtn_Click()
    Form1.Close
END
```

One additional change must be added to the Quit button. Click on the Quit button in the form, and the Properties tab will appear. Find the **Cancel** property, and change it from False to True. This will allow the **ESC** key to close the program, just as if you had clicked on the Quit button.

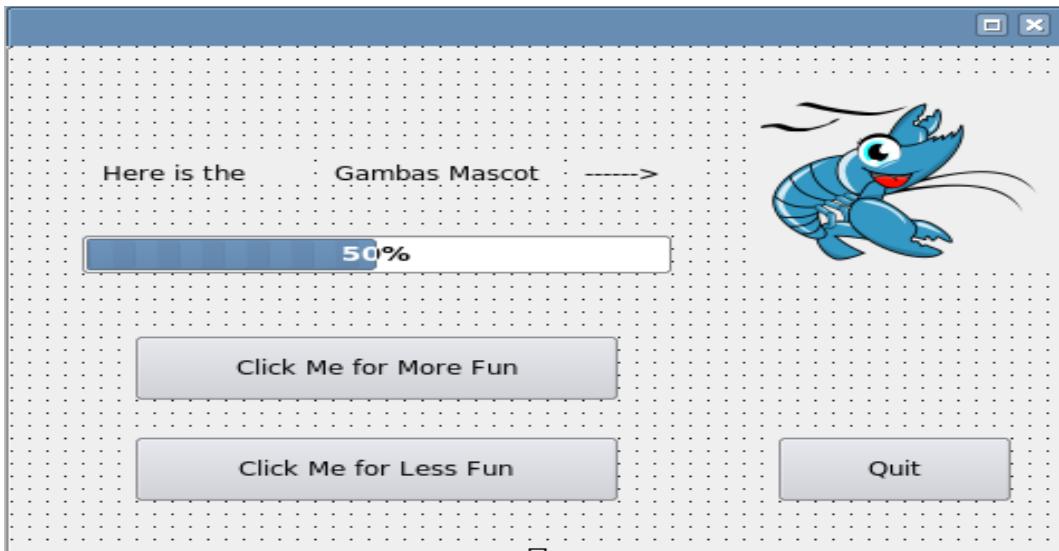


Figure 26: Adding the FunBtn to our form.

Now, we have to write code for the ProgressBar. Note that at this time of writing, the ProgressBar in GTK+ does not work correctly – one example of why we have a preference for using Qt in Gambas. For our example, we will increment the ProgressBar when the FunBtn is clicked and decrement it when the NoFunBtn is clicked.

Double-click on the FunBtn control and add this code:

```
PUBLIC SUB FunBtn_Click()
    ProgressBar1.Value = ProgressBar1.Value + 0.01
    IF ProgressBar1.Value > 0.99 THEN
        ProgressBar1.Value = 0.0
    ENDIF
END
```

A Beginner's Guide to Gambas – Revised Edition

Next, double-click on the NoFunBtn and add this:

```
PUBLIC SUB NoFunBtn_Click()
    ProgressBar1.Value = ProgressBar1.Value - 0.01
    IF ProgressBar1.Value < 0.01 THEN
        ProgressBar1.Value = 0.0
    ENDIF
END
```

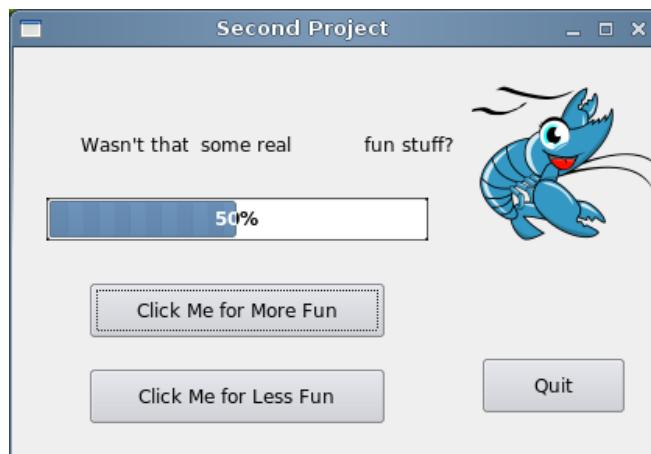


Figure 27: Initial display of our form.

That is nearly all there is left to do! Now, save the project and click the run button. You should see something similar to the figure above on program start. Now, click the “Click me for more fun” button to see that the progress bar will change. Click it three times and you should see this:

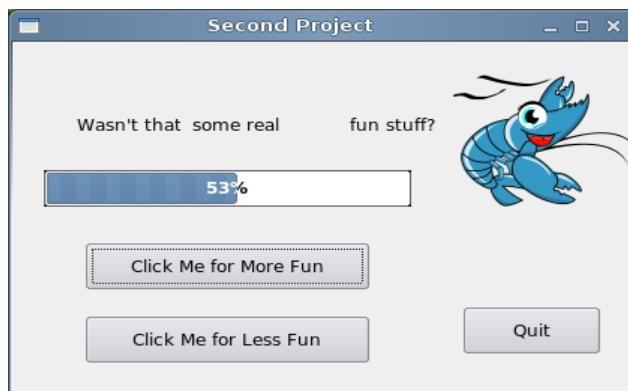


Figure 28: Figure 10: The progress bar when the FunBtn is clicked three times.

Note the text field disappears when the mouse enters the form. If you move your mouse over the mascot picture, you should see the label text appear and disappear as you move the

mouse on and off the picture. Finally, you can hit ESC or click the Quit button to exit the program. The last event we are going to write code for in this example is the **double-click** event. This requires you to once again choose an *event handler*.

From the Gambas IDE, move the mouse over the FunBtn and click it once. Next, right click the mouse and you will see the menu pop-up allowing you to choose Events. From the Events submenu, pick the **Dblclick** event. Add this code, as shown below:

```
PUBLIC SUB FunBtn_DblClick()
ProgressBar1.Value = ProgressBar1.Value + 0.1
IF ProgressBar1.Value > 0.90 THEN
    ProgressBar1.Value = 0.0
ENDIF
END
```

Repeat this process for the NoFunBtn and add this code:

```
PUBLIC SUB NoFunBtn_DblClick()
ProgressBar1.Value = ProgressBar1.Value - 0.1
IF ProgressBar1.Value < 0.1 THEN
    ProgressBar1.Value = 0.0
ENDIF
END
```

Now, save the project and click the run button. This time, when you click the progress bar it works just as it did before, but if you double-click, it will increment the value by 10 units rather than by one. Once you exit your program, save your project and exit Gambas. Next, we are going to look at the other events that can be used with the Button control.

Button Events

The Button events we have used in our previous example include the Click, DblClick, Enter, and Leave events. We will reserve our discussion of the remaining events, namely Drag, DragMove, Drop, LostFocus and GotFocus for when we deal with drag and drop operations. The Menu, KeyPress and KeyRelease events will be covered when we talk about menu and keyboard operations. Finally, when we cover mouse operations, we will discuss the MouseDown, MouseMove, MouseUp, and MouseWheel events.

The Picture Class

Earlier in the chapter, when we used the Picture control to place our mascot on the form in our second example, we mentioned the Picture Class and said we would cover it later. Well, now it is time to talk about the Picture Class. This class represents a picture and its contents are stored in the display server, not in process memory like an Image would be

normally stored. Even if X-Windows does not manage transparency (yet), each picture object can have a mask assigned with it. This can be set explicitly at the time the picture is instantiated, or it can be set implicitly when loading an image file that has transparency attributes, like a PNG file. When drawing on a picture having a mask, both the picture and the mask are modified. This class is creatable. It is defined as:

```
DIM hPicture AS Picture
```

and the process of instantiating it uses this format:

```
hPicture = NEW Picture ([Width AS Integer, Height AS Integer, Transparent AS Boolean ])
```

The code above will create a new picture object. If the Width and Height are not specified, the new picture is empty (void). You can specify if the picture has a mask with the Transparent parameter. The picture class acts like an array. For example:

```
DIM hPicture AS Picture
```

```
hPicture = Picture [ Path AS String ]
```

will return a Picture object from the internal picture cache. If the picture is not present in the cache, it is automatically loaded from the specified file. To insert a picture in the cache, use this call:

```
Picture [ Path AS String ] = hPicture
```

Properties for this control include Depth, Height, Image, Transparent, and Width. The methods provided include Clear, Copy, Fill, Flush, Load, Resize, and Save. We will cover the use of these methods and properties in later examples in this book. At this point, you should have a solid understanding of how to use the Gambas programming environment and its various tools. In the next few chapters, we will present more controls with more complex examples as the best way to learn is by doing.



Chapter 5 – Controls for User Input

In our introduction to controls, we learned about the most basic types of controls and events, namely pictures, labels, buttons, and a progress bar. We also learned how to detect when the mouse is over a control and what actions we can take when it moves on or off the control. These types of controls all present data or respond to events like the mouse or a button click, but they do not allow users to choose from among alternatives such as picking items from a list or typing in their name, etc. In this chapter, we will add to our arsenal of Gambas knowledge by learning how to accomplish those tasks and respond to even more events. We are going to build a program that will introduce you to the following controls and teach you how to use them:

- ✓ TextLabel
- ✓ TextBox
- ✓ ComboBox
- ✓ ListBox
- ✓ ToggleButton
- ✓ Panel
- ✓ Frame
- ✓ Checkbox
- ✓ RadioButton

Here is what our program will look like when we are finished with it:

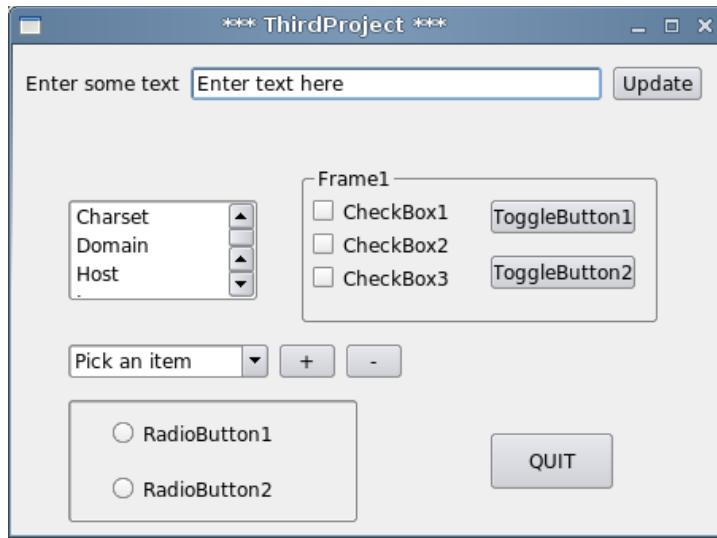


Figure 29: Third Project final results.

In this exercise, we will learn to put titles on our program window, use an input box to gather HTML-formatted input from the user and display that result in a new type of label, the **TextLabel**. We will also learn how to group and organize our controls on the form, using

panels and frames, and place controls like **RadioButtons** and **CheckBoxes** within those organizational controls. Finally, we will learn how to use **ListBoxes** and **ComboBoxes** to allow the user to select from one of many choices. Along the way, we will learn a bit more about event coding and try to make it interesting.

Let's get started by creating a new Project with Gambas. Start Gambas and choose to create a new project using a QT graphical application. Go through the wizard making the project translatable and the form controls public. Select the name **ThirdProjectQT** and put it in a directory of your choosing. When the IDE starts, create a new, startup class form by double-clicking on the FMain form icon. Double-click on the blank form, and the code editor window will appear. You should see something like this:

```
' Gambas class file
PUBLIC SUB Form_Open()
END
```

Now, we will start by putting the title of our program at the top of the window when the program runs. To accomplish this, we will set the form's **Caption** property to the text we want to use as the title. Enter this line in the **Form_Open()** subroutine:

```
ME.Caption = " *** Third Project *** "
```

That takes care of the text for the window title. Now, if we look at our final result picture, we still have to add our controls. We will start at the top, adding the controls we already know about: **Label**, **TextBox**, and **Button**. Place each control on the form as shown in Figure 26 above. Name the controls *Label1*, *TextBox1*, and *UpdateBtn*. These controls were used in the last chapter, so you should be comfortable using them in your programs. Next, we will add the Quit Button at the bottom right corner of the form. Name this control *QuitBtn*. Now, let's add the code for these controls. Double-click the mouse on the *QuitBtn* and add this code to the **QuitBtn_Click()** subroutine:

```
PUBLIC SUB QuitBtn_Click()
  FMain.Close
END
```

For the update button, we are going to transfer the text the user typed into the **TextBox** control to the **TextLabel**, reflecting any HTML formatting that they entered. Here is the code needed for that:

```
PUBLIC SUB UpdateBtn_Click()
  TextLabel1.Text = TextBox1.Text
END
```

From this point on, we will be learning to use new controls and features of Gambas. We have already written code to assign input data to the `TextLabel`, so let's take the time now to learn all we can about it. Before we start, it is probably a good idea to go ahead and click the save button on the IDE to save your work.

TextLabel

This class is creatable and it inherits its attributes from the `Control` class. Like all of the controls that contain text, Gambas allows you to set the formatting in the Properties tab. In `TextLabel`, you can set: Alignment (Left, Center, or Right); Font (Typeface, Size, Bold, Underline, Strikeout); and both Background and Foreground colors and shading. `TextLabel` also implements a control capable of displaying simple HTML text, -- this is useful when creating and formatting controls that contain text in your code for classes (i.e., when you are doing it the “hard way” – not using a form to create these objects). You can use HTML entities such as `<` and `>` to display characters like `<` and `>`. For HTML that requires the use of quote marks, avoid quotes(“..”) because it will not work. For example, do not enter:

```
<div align="center"> <b> This won't work.</b>
```

Type this instead:

```
<div align=center><b> This will work.</b>
```

The figure below shows you an example of how typing HTML in the `TextLabel` box works:



Figure 30: Using HTML to format a text label.

Of course, no one expects the application user to format their own text when typing in a `TextBox`, using HTML. It is far more useful to write the HTML formatting into your string-handling code for `TextLabel`. This way, Gambas lets us include formatting effects that are not otherwise available, unless you created the `TextLabel` object on a form. Here are a few of the HTML tags that will work in your `TextLabel` code (and when you include them in code, they must be enclosed in quotes (“..”)):

alignment: “`<div align=center>`”

(or you may also use “**left**” or “**right**” in place of the word “**center**”)

bold: “`..`”

A Beginner's Guide to Gambas – Revised Edition

```
underline: "<u>..</u>"  
strikethrough: "<s>..</s>"
```

In HTML, when you combine tags, the start tags and end tags must be in reverse order. For example:

```
"<div align=center><u><b>"..."</b></u>"
```

One text attribute that you cannot control in TextLabel code is color:

```
"<FONT COLOR = FF0000><b>" & "This will NOT work." & "<b></FONT>"
```

TextLabel is defined as **DIM hTextLabel AS TextLabel** and you declare it like this:

```
hTextLabel = NEW TextLabel ( Parent AS Container )
```

What we want to do now is code the update button to take advantage of these HTML features. To accomplish this, we need to modify the code we previously entered for the update button's click event. Double-click on the UpdateBtn control and change the code to look like this:

```
PUBLIC SUB UpdateBtn_Click()  
TextLabel1.Text= "<div align=center><u><b>" & TextBox1.Text & "</u></b>"  
END
```

In addition, change the code in the TextBox1_Leave event to look like this:

```
PUBLIC SUB TextBox1_Leave()  
TextLabel1.Text = "<div align=center><u><b>" & TextBox1.Text & "</b></u>"  
TextBox1.Text = "Enter text here"  
END
```

Instead of simply assigning the input text from the TextBox to the TextLabel1.Text property, we are going to make it look fancy. Now, anything the user types will automatically be centered, underlined, in boldface and placed on the TextLabel1. If you save your project and run it at this time, your results should be like that of the Figure below:



Figure 31: Modified TextLabel output using HTML formatting.

TextBox

Our next task is to learn more about the TextBox control we just used. You already know that it is used to let the user enter a line of text input and return that input to the program as a string. TextBox inherits its attributes from the Control class and this class implements a single line text edit control. This class is creatable and is declared as:

DIM hTextBox AS TextBox

To create and instantiate a new TextBox control in your code, you can use this format:

hTextBox = NEW TextBox (Parent AS Container)

Let's try something else. Suppose that we want to streamline our application so that the user does not have to click the update button when they enter text. Also, it would be nice to have the text field clear itself if the mouse entered the control area and to put back the default prompt when when the mouse is moved away from the control. We would want to preserve whatever was typed into the field by the user when the mouse leaves the control so we have to remember to assign the value that is currently in the TextBox1.Text field to the TextLabel1.Text field. We want to continue to use the HTML-capable format capabilities of the TextLabel.

We will need to modify our code. First, we must create the event handlers for the mouse entering the control and leaving the control. Click once on the TextBox1 control on the form and right-click the mouse to select the Event item. From the Event submenu choose **Enter**. Repeat this process and choose the **Leave** event. Now, go to the code editor, find the first subroutine named **TextBox1_Enter()** and insert the following code between the first and last line:

TextBox1.Clear

That's it for this routine. We just call the built-in method to clear the TextBox and we are done. Now, we still need to go to the **TextBox1_Leave()** subroutine and enter this code:

```
TextLabel1.Text = "<div align=center> <b>" & TextBox1.Text & "</b>"  
TextBox1.Text = "<Enter text here>"
```

The first line of code will use HTML to format and center the TextBox1.Text string and set it to a boldface font. The next line of code assigns the default prompt string we initially created to the TextBox1.Text so when the mouse leaves we have saved whatever the user typed and reset it to what it looked like before the mouse entered the control. Save your work and run the program to see how it works.

There are not really any TextBox-unique Methods to discuss, but while we are learning

about events, let's play around with the code a little and learn about the Show and Hide methods and how they may be used in a program. Click once on the form (be careful not to click on any control on the form) and right-click the mouse to select the Event item. From the **Event** submenu choose **MouseDown**. Repeat this process and choose the **DblClick** event. Now, go to the code editor, find the subroutine **Form_MouseDown()** and insert the following code between the first and last line:

[UpdateBtn.Hide](#)

That is all we need to do to hide the UpdateBtn control. If you user opts not to use the button now, all the need to do is click the form and it will hide the button. How do we bring it back? Double-click will do the trick nicely. From the code editor, find the **Form_DblClick()** subroutine and enter this code:

[UpdateBtn.Show](#)

Right now, you may be asking yourself “*How would the user ever know this?*” Good point. Let's let them know with a ToolTip property. Click on the UpdateBtn control. Now go to the property window and find ToolTip. On the right side of that entry is an input field and with a gray box with three dots (...). That means another dialog will take place when you click. In this case, it will open an edit window so you can format your ToolTip using HTML. Click it and when the edit dialog appears, enter this text (or code, as you wish):

[Click \(or double-click\) on the form to hide \(or show\) me](#)

Now, save your work and run the program to see how it works. Hover the mouse over the Update Button and the nicely formatted ToolTip will appear. It should look like the Figure below. Cool, huh?



Figure 32: Adding a ToolTip to inform the user how to show/hide a control.

There are a couple of TextBox-unique events we could discuss at this point, namely **KeyPress** and **KeyRelease**, but we are going to hold off on those until later in the book when we start working with the keyboard directly. Ready to move on? We are going to add some selector controls to our program next. The ComboBox and ListBox allow users to select from among several choices. We will start with the ComboBox first.

ComboBox

The ComboBox control inherits its attributes from the Control class. It implements a text box combined with a popup list box. This class is creatable and is declared using the format below:

```
DIM hComboBox AS ComboBox
```

To instantiate the variable we just declared, use this format:

```
hComboBox = NEW ComboBox ( Parent AS Container )
```

This will create a new ComboBox control. This class acts like a read-only array. In other words, declaring the variable, instantiating it, and retrieving a value from code directly would work like this:

```
DIM hComboBox AS ComboBox  
DIM hComboBoxItem AS .ComboBoxItem  
DIM Index as Integer  
hComboBoxItem = hComboBox[Index]
```

The line of code above will return a combo-box item from its integer index. Note that the **.ComboBoxItem** represents an item in the combo-box popup list-box. This class is virtual. You cannot use it as a data-type and it is not creatable. It has a single property, Text which returns a string value representing the ComboBox item represented by Index. We are going to create a ComboBox on our form now. Here is what our control will look like:

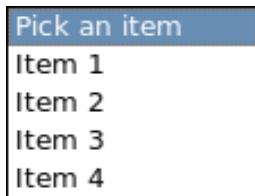


Figure 33: Our
ComboBox.

OK, i agree it is not very fancy and it probably ranks kind of low on the creativity list, but it will get the job done and teach you what you need to know to use ComboBoxes. To build our new control, we will go to the Gambas ToolBox and select the ComboBox control. Place it on the form (*refer to our initial Figure at the very beginning of this chapter to see where it should go*) and try to get it as close to what you see in that picture as you can. Once you are done with that, click on it once if you do not see the handles and go to the Properties Window.

A Beginner's Guide to Gambas – Revised Edition

Find the List property. The input box to the right of this will enable a selector dialog if you click inside it. Click the selector dialog button (remember, with the three dots?) and it will bring up the Edit list property Dialog shown in the figure above.

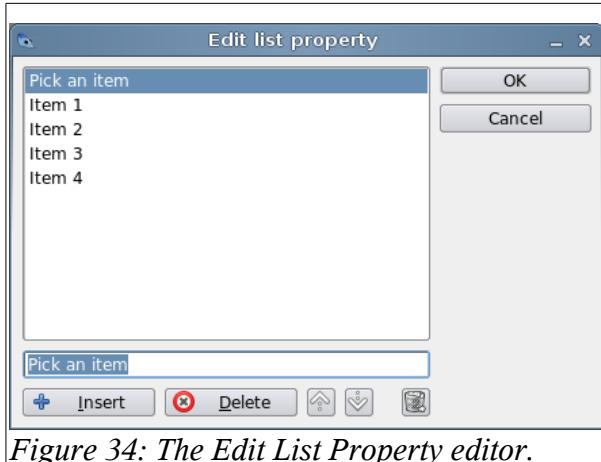


Figure 34: The Edit List Property editor.

To insert an item in the list, type the item in the TextBox at the bottom of the editor. The first item in the list is the default. It is most often used to set a default selection (like *None* or *Pick something*, etc.) In our case, type Pick an item and click insert. Add Item1, Item2, Item3, and Item 4 to the list. Make sure they appear in the order shown in the Figure above so it will look right when it runs. Once you have it set the way you want, just click the OK button and the editor will close and save your work. Save your project and run the program to see what it looks like on your form. When you have finished looking at the work we have done so far, exit the program and we will continue by adding some code to process ComboBox selections.

If the user selects something from the ComboBox, the item will change and this change generates an event that you can handle. From the form, select the ComboBox control and right-click. Choose Event and select Change from the submenu. Now, go to the code editor and find the subroutine **PUBLIC SUB ComboBox1_Change()**. Add the code so the subroutine looks like this:

```
PUBLIC SUB ComboBox1_Change()
DIM result AS String
result = Trim(ComboBox1.Text)
TextLabel1.Text = result & " was selected from the ComboBox"
END
```

Save your project and run the program to see what happens. Notice that when an item is selected the TextLabel1.Text value is updated but does not have any fancy formatting that it is capable of. Well, that is easy to fix. Let's change the code like this:

```
PUBLIC SUB ComboBox1_Change()
```

```
DIM result AS String
result = "<div align=center> <b>" & Trim(ComboBox1.Text) & "</b>"
TextLabel1.Text = result & " was selected."
END
```

Change the code and run the program again. You should see something similar to this in the TextLabel1 control:



Figure 35: Formatting a TextLabel with HTML.

So the real programmers out there are already asking how to do this with code, eh? I thought you would never ask! Seriously, it is not hard. Let's modify our program slightly to allow dynamically add or remove items from the ComboBox. To do this, we are going to need a button that will indicate adding an item and another button to remove an item. Add two buttons, one called PlusBtn and the other called MinusBtn to the form, as shown below:



Figure 36: Using plus and minus buttons to alter the ComboBox list.

Double-click on the PlusBtn and insert this code:

```
PUBLIC SUB PlusBtn_Click()
DIM Item AS String
DIM NumItems AS Integer
ComboBox1.Refresh
NumItems = ComboBox1.Count
IF NumItems = 0 THEN
    Item = "Pick an Item"
ELSE
    Item = "Item " & Str(NumItems)
ENDIF
ComboBox1.Refresh
ComboBox1.Add(Item,NumItems)
END
```

In the code above, we declare two variables, *Item* and *NumItems*. *Item* is a string and *NumItems* an Integer. We will use *NumItems* to determine how many items are already in the list. This is done using the **.Count** property. If there are no items in the list, we will set our default prompt of “Pick an Item” as Item zero.

ComboBox arrays are zero-based, meaning the counting starts at zero, not one. If the NumItems value is zero we will create a default string. Otherwise, we are going to concatenate the item number to the word Item so it will add to the next open slot in the list. We call the Refresh method to force the ComboBox to refresh its count property so if the user tries to select again it will be current. The last line of code uses the .Add method to add the item to the ComboBox item list. Now, double-click on the MinusBtn and add this code:

```
PUBLIC SUB MinusBtn_Click()
    DIM NumItems AS Integer
    'The combobox array starts at zero
    ComboBox1.Refresh
    NumItems = ComboBox1.Count
    IF NumItems > 0 THEN
        DEC NumItems
        IF NumItems <> 0 THEN
            ComboBox1.Remove(NumItems)
        ENDIF
        ComboBox1.Refresh
    ENDIF
END
```

Removing elements from the list is much simpler. We only need an integer variable to determine the number of elements in the list. Once again, this is done using the .Count property. In our example, Count will return a value of 5 the first time it is read. If there are no items in the list, we will not do anything. Because the ComboBox items are stored in a zero-based array, we must decrement the count by one if it is not already zero. Now, save your project and run the program. Try removing all the items in the list and re-inserting them. Cool, eh? More cool stuff to come with the ListBox, which we discuss next.

ListBox

The ListBox control also inherits its attributes from the Control class. The ListBox implements a list of selectable text items. This class is creatable. The following code will declare a ListBox and create it:

```
DIM hListBox AS ListBox
hListBox = NEW ListBox ( Parent AS Container )
```

Like the ComboBox, this class acts like a read-only array.

```
DIM Index as Integer
DIM hListBox AS ListBox
DIM hListBoxItem AS .ListBoxItem
```

hListBoxItem = hListBox[Index]

The line of code above will return a ListBox item from its integer index. Note that the **.ListBoxItem** represents an item in the ListBox popup list-box. This class is virtual. You cannot use it as a data-type and it is not creatable. It has a single property, Text which returns a string value representing the ListBox item represented by Index. We are going to create a ListBox on our form now. Here is what our new control will look like:



Figure 37: What our
ListBox will look like.

In order to build our new control, we will go to the Gambas ToolBox and select the ListBox control. Place it on the form (*refer to our initial picture at the very beginning of this chapter to see where it should go*) and try to get it as close to what you see in that picture as you can. Once you are done with that, click on it once if you do not see the handles and go to the Properties Window. Find the List property. The input box to the right of this will enable a selector dialog if you click inside it. Click the selector dialog button (remember, with the three dots?) and it will bring up the Edit list property Dialog (shown below).

This is exactly the same editor we used to build the ComboBox1 control. This time, we are going to add items to the list and also learn about the System class. The System class is part of the Gambas **component** library and provides support for all the classes included in the interpreter by default.

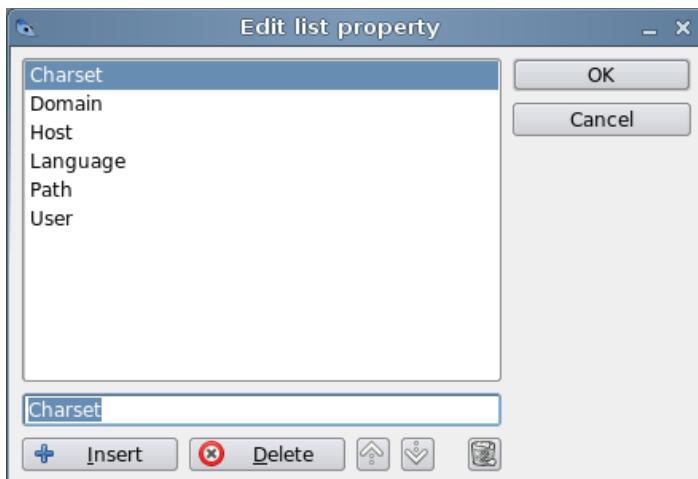


Figure 38: The ListBox edit list property editor.

The System class is static and provides read-only information about the operating system environment. The properties of this class that you can read include Charset, Domain, Home, Host, Language, Path, and User. Use the editor and enter the following items in the list:

- ✓ Charset
- ✓ Domain
- ✓ Host
- ✓ Language
- ✓ Path
- ✓ User

Next, double-click on the ListBox1 control on the form. This will take you to the code editor and you should be in the **PUBLIC SUB ListBox1_Click()** subroutine. The following code will show you how to access these values. Enter it as follows:

```
PUBLIC SUB ListBox1_Click()
IF Trim(ListBox1.Text) = "System.Charset" THEN
    TextLabel1.Text = System.Charset
ELSE IF Trim(ListBox1.Text) = "Domain" THEN
    TextLabel1.Text = System.Domain
ELSE IF Trim(ListBox1.Text) = "Host" THEN
    TextLabel1.Text = System.Host
ELSE IF Trim(ListBox1.Text) = "Language" THEN
    TextLabel1.Text = System.Language
ELSE IF Trim(ListBox1.Text) = "Path" THEN
    TextLabel1.Text = System.Path
ELSE IF Trim(ListBox1.Text) = "User" THEN
    TextLabel1.Text = User.Home
ENDIF
END
```

Now, save your project and run the program. Try selecting all the items in the list and see what information is returned. Lists and ComboBoxes are very easy to implement and use. A great advantage of using these types of selector components is that it virtually eliminates the possibility of a user typing data incorrectly. In the next section, we are going to learn how to organize the controls on our form to better manage the presentation to controls to the user. Also, we will learn why frames and panels are a good thing to consider early in the development process.

Frame

The Frame class inherits its attributes from the Container class. This control is a container with an etched border and a label. This class is creatable. Do declare and

instantiate a Frame, use this format:

```
DIM hFrame AS Frame  
hFrame = NEW Frame (Parent AS Container)
```

The great thing about a frame is that it works sort of like a window within a window. Any control you place in the frame becomes a part of the frame, in a manner of speaking. What this means is that if you put CheckBox controls or buttons or whatever in the frame and decide to move the frame, they all move with it. They can be rearranged within the frame, but if the frame moves, hides, etc., so do they. Here is what our frame and the controls we will add to it look like:

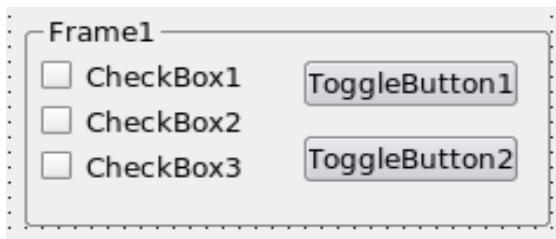


Figure 39: What the example frame we build will look like.

To continue development of our project, we will first add a **Frame** control to the form (name it Frame1) and place it similar to that found in the Figure at the beginning of this chapter. Next, we will add two **ToggleButton** controls (named ToggleButton1 and ToggleButton2) and three **CheckBox** controls (named CheckBox1, CheckBox2, and CheckBox3). Arrange them to look like the figure above. Once you have everything placed correctly on the form, let's go to the next step and write code for the events we will respond to in our program.

ToggleButton

The ToggleButton control inherits its attributes from the Control class and implements a toggle button. This means it either toggles up or down. This class is creatable. To declare and instantiate this control, use the format:

```
DIM hToggleButton AS ToggleButton  
hToggleButton = NEW ToggleButton ( Parent AS Container )
```

Double-click on the ToggleButton1 control and add this code to the **PUBLIC SUB ToggleButton1_Click()** subroutine:

```
PUBLIC SUB ToggleButton1_Click()  
IF ToggleButton1.Value = TRUE THEN
```

```
Frame1.Text = "Toggled 1 down"
ELSE
    Frame1.Text = "Toggled 1 up"
ENDIF
END
```

Repeat the previous step for the ToggleButton2 control and enter this code:

```
PUBLIC SUB ToggleButton2_Click()
IF ToggleButton2.Value = TRUE THEN
    Frame1.Text = "Toggled 2 down"
ELSE
    Frame1.Text = "Toggled 2 up"
ENDIF
END
```

What we have done with the code above is check the ToggleButton.Value property to see if it is TRUE, indicating that the button was clicked down. If FALSE, another click toggled it to the up position. Regardless of the position it is at, we want the Frame1.Text to display the status of the button last clicked. Now, let's move on to learn about the **CheckBox** controls.

Checkbox

The CheckBox class inherits its attributes from the Control class. This class implements a check-box control and it is creatable. Declare and instantiate the variable like this:

```
DIM hCheckBox AS CheckBox
hCheckBox = NEW CheckBox ( Parent AS Container )
```

When a CheckBox control is clicked in our program, we want to detect the click event and immediately respond. In this case, we will change the Checkbox.Text property whenever the CheckBox.Value is checked and turns out to be TRUE. This will show that we caught the click event and responded to the value of TRUE or FALSE (checked or unchecked). If the box is unchecked, we will want to return it to the “Normal” state. For CheckBox1, double-click on the control and enter this code in the code editor for the **PUBLIC SUB CheckBox1_Click()** subroutine:

```
PUBLIC SUB CheckBox1_Click()
IF CheckBox1.Value = TRUE THEN
    Checkbox1.Text = "I was picked"
ELSE
    Checkbox1.Text = "Checkbox1"
```

```
ENDIF  
END
```

Repeat this process for **CheckBox2** and **CheckBox3**:

```
PUBLIC SUB CheckBox2_Click()  
IF CheckBox2.Value = TRUE THEN  
    Checkbox2.Text = "I was picked"  
ELSE  
    Checkbox2.Text = "Checkbox2"  
ENDIF  
END
```

```
PUBLIC SUB CheckBox3_Click()  
IF CheckBox3.Value = TRUE THEN  
    Checkbox3.Text = "I was picked"  
ELSE  
    Checkbox3.Text = "Checkbox3"  
ENDIF  
END
```

Now, save your project and run the program. Once you have satisfied yourself that they work as we planned, end the program and we will continue our project by adding the last two controls that we are going to cover in this chapter, the **Panel** and **RadioButton**.

Panel

The Panel class inherits its attributes from the Container class. This class implements a Panel control with a changeable border. This class is creatable. Declare and instantiate a panel like this:

```
DIM hPanel AS Panel  
hPanel = NEW Panel ( Parent AS Container )
```

We will first need to add a Panel control to the form (name it Panel1) and place it to look like the Panel found at the beginning of this chapter. Next, we will add two RadioButton controls (named RadioButton1 and RadioButton2). Arrange them to look like the figure shown below. In our program, we are going to use the Panel to group our RadioButtons inside it. Here is what this panel will look like when we are finished:



Figure 40: A Panel control containing RadioButtons.

RadioButton

The RadioButton class inherits its attributes from the Control class and is used to implement a radio button control. RadioButton controls that share the same parent (in this case, the Panel container) are mutually exclusive. Only one RadioButton can be selected at once. The RadioButton class is creatable so you can declare and instantiate the RadioButton like this:

```
DIM hRadioButton AS RadioButton  
hRadioButton = NEW RadioButton ( Parent AS Container )
```

Select the RadioButton control from the ToolBox and place it on the form. Name the first RadioButton RadioButton1 and repeat this process for another, named RadioButton2. Once you have everything placed correctly on the form, let's go to the next step and write code. Double-click on the first RadioButton and enter this code:

```
PUBLIC SUB RadioButton1_Click()  
    RadioButton1.Text = "HaHa RB2"  
    RadioButton2.Text = "RadioButton2"  
END
```

Repeat this process for the second one:

```
PUBLIC SUB RadioButton2_Click()  
    RadioButton2.Text = "HaHa RBI"  
    RadioButton1.Text = "RadioButton1"  
END
```

Now, save your project and run the program. It should resemble the figure below:

A Beginner's Guide to Gambas – Revised Edition

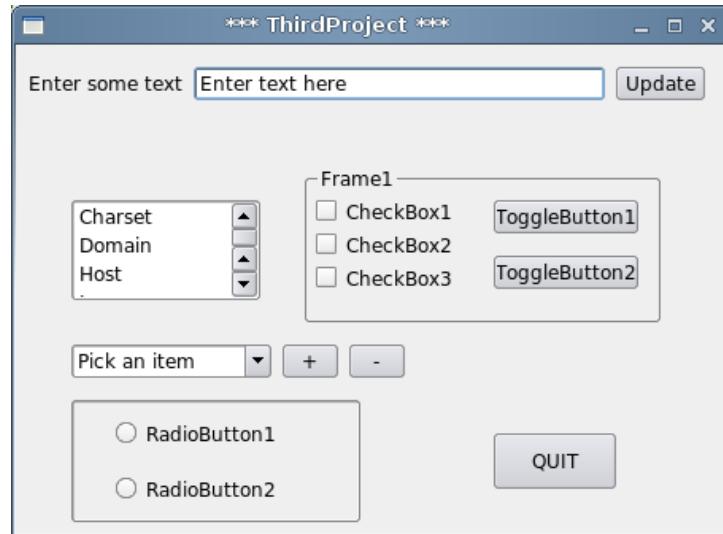


Figure 41: Third Project final results.

Once you have satisfied yourself that everything we have done in this chapter will work as we planned, take a well deserved break and we will start fresh on the next chapter.



Chapter 6 – Menus, Modules, Dialogs and Message Boxes

Almost every GUI-based program you encounter will use some combination of menus, MessageBoxes, and standard dialogs to communicate with the user. Gambas is not any different. Additionally, there is no programming language that provides everything that a programmer would use as a pre-built component. That is what modules are used for – writing what you need outside the scope of what Gambas provides. For example, if you need a function that returns the name of a color based on what is chosen in a standard color selection dialog, you are going to have to write one, and we will do just that in our next sample project. In this chapter, we will learn how to use each of these controls and build another example program to help you learn to master Gambas. Here is a screenshot of the final version of what our project will look like:

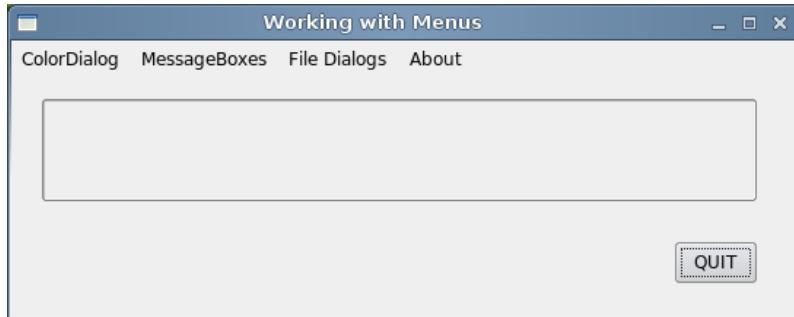


Figure 42: Menu project final result.

We are going to create four menus. Each menu will be used to demonstrate the controls or standard dialogs we are going to learn about in this chapter. We will first build the menu, as shown above, then we will show you how to use the standard color and file dialogs, MessageBoxes, and, as an added bonus, we will show you how to create a module in Gambas and use it. To begin our project, open Gambas and from the wizard, in the “Type” column, select “QT graphical application”. Name it MenuProject and check “Internationalization”. Once you get to the IDE, double-click on FMain.

Double-click on the form anywhere outside a control boundary and you will get the FMain.class page. Immediately after '*Gambas class file*', type

ColorName AS String

This dimensions the global variable ColorName.

The declaration of the **PUBLIC SUB Form_Open()** subroutine appeared automatically in the code editor. Insert this code in that empty subroutine:

ME.Caption = " Working with Menus "

Add a Quit Button to the form, as shown above and enter this code to be executed when a user clicks the Quit button:

```
PUBLIC SUB QuitBtn_Click()
FMain.Close
END
```

Next, add a TextLabel, named TextLabel1 to the window as shown in the Figure above. That takes care of the preliminaries for our project. Now, let's see how to build the menu and use the standard dialogs provided by Gambas.

The Gambas Menu Editor



Figure 43: The Gambas Menu Editor when it starts.

Gambas has a built-in Menu Editor. You can access it from the form window by pressing **CTRL-E** or by right-clicking on the form and selecting the Menu Editor option from the popup that appears. Either way, the Menu Editor will appear as shown above.

When you first start the Menu Editor, the editor window will, of course, be blank. It is not an intuitively obvious tool to use, so let's spend a minute to explain the Menu Editor. First, some common terminology. The text that you see running horizontally across the top of a form is referred to as a menu.

A menu can contain **MenuItems** (which are listed vertically below the menu).

MenuItems can contain subMenus, which are basically new nested menus that belong to a MenuItem and will popup and display another list of MenuItems. Menus can contain several layers of nested subMenus.

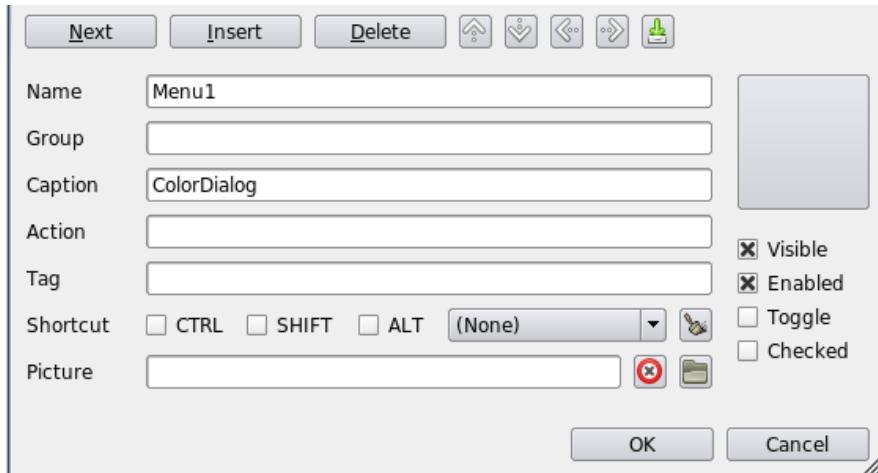


Figure 44: Editing menu entries in the Menu Editor.

To insert a menu you will have to use the **Insert** button. The **Next** button will move the current menu cursor position to the next item in the window. It only moves down and there is no previous button. If you reach the end of the list, it will stop and the only thing you can do is click the mouse on a menu entry or insert a new one. The **Delete** button will remove the menu entry under the menu cursor.

The first two \uparrow and \downarrow arrow buttons (up and down) to the right of the Delete button will move the entry under the menu cursor up or down the list of menu entries. If you click the \leftarrow or \rightarrow (un-indent or indent) arrow buttons, the process will work much like indentation in an outline works. The indented item will become either a MenuItem or subMenuItem depending on the level of indentation.

Indented items with an ellipsis (three ...) preceding the name indicate MenuItems. More than three dots preceding the name indicates it is a subMenuItem of the entry at the previous indentation level. Once you build a menu or two, it will be easy to understand.

When you click the insert button, you will see several new fields appear at the bottom of the Menu Editor; The **Name** field should probably have been called *Variable Name* because that is what this field is used for. This is the variable name that your code in the class file will use to reference this particular object.

The **Group** field refers to a **Control Group**. Control Groups are used in the Gambas IDE to create and handle groups of controls as a single entity. When you select any control on a form, in the Properties dialog, you should notice a Group property that doesn't correspond to any documented property of that control. This special *pseudo-property* allows you to assign control groups. Essentially, when that control is created at run-time, it will be treated as

though you had created it in code like so:

```
myControl = NEW ColumnView(ME) AS "myGroup"
```

and its event handlers must be named like so:

```
PUBLIC SUB myGroup_Click()
```

You can refer to the control that generated the event with the **LAST** keyword, or you can assign a **Tag** to each control to differentiate them by using the tag value.

Caption is the text that the end-user will see when the program runs. **Tag** is reserved for programmers to store data and is a common property for nearly all controls. We will cover how to use the Tag property later. For now, you only need to know that it can contain ANY Variant type data.

The **Shortcut** CheckBoxes allow you to define keyboard shortcut key combinations like Control-A to automatically invoke the click event for that menu entry. Remember, menu entries basically only respond to click events or keyboard shortcuts. The **Picture** field allows you to specify the file name of an icon that will display with the menu entry. A preview window (a.k.a., a picture box) appears to the right of this field.

The CheckBoxes to the right of the input fields, **Visible**, **Enabled**, and **Checked** refer to the *state* of the menu entry. By default, all menu entries are created to be enabled and visible. You can specify if the item is checked or not. In code, you can see if the entry is checked or not by looking at the menu entry's **Checked** property. It will return a TRUE or FALSE value indicating the status of the item. Here is an example of how to toggle a checkmark on or off when a menu item is picked:

```
IF Menu2Item1.Checked THEN  
    Menu2Item1.Checked = FALSE  
ELSE  
    Menu2Item1.Checked = TRUE  
ENDIF
```

The IF statement will test the Checked property of the Menu2Item1 object and if the Boolean value is TRUE will set it to FALSE (toggling it). If the Boolean value is NOT TRUE the ELSE clause is executed, toggling the .Checked value to TRUE. This logic can be used to toggle any menu or subMenuItem.

A word of caution – use this feature sparingly for good GUI design. Many users feel overwhelmed when they are presented with too much gadgetry in an interface.

User interface design has been a topic of considerable research, including on its aesthetics. In the past, standards have been developed, as far back as the eighties, for defining the usability of software products. The International Federation for Information Processing (IFIP) user interface reference model is often used in UI design. The model proposes four dimensions to structure the user interface:

- ✓ The input/output dimension (the look)
- ✓ The dialogue dimension (the feel)
- ✓ The technical or functional dimension (the access to tools and services)
- ✓ The organizational dimension (the communication and co-operation support)

This model has greatly influenced the development of the international standard ISO 9241 describing the interface design requirements for usability. A fundamental reality of application development is that the user interface is the primary system to the users. What users need is for developers to build applications that meet their needs and that are easy to use.

Too many developers think that they are artistic geniuses – they do not bother to follow user interface design standards or invest the effort to make their applications usable, instead they mistakenly believe that the important thing is to make the code clever or to use a really interesting color scheme. Not so! User interface design is important for several reasons. First of all, the more intuitive the user interface, the easier it is to use, and the easier it is to use, the less expensive it is to support it. A good user interface makes it easier it is to train people to use it, reducing your training costs. The better your user interface the more your users will like to use it, increasing their satisfaction with the work that you have done.

Building Menus

Now, let's create our menu. When you are finished, it will look something like the figure shown below which is a screenshot taken from the menu editor. The Menu Editor is available only when a form is being designed.

The first step in creating a menu is to enter the menu item's [variable] name and its caption to be displayed to the user. The caption can incorporate the ampersand (&) for an access key designation, also known as an accelerator key. This enables the user to see an underlined letter in the menu and use the Alt key along with the accelerator key.

A Beginner's Guide to Gambas – Revised Edition

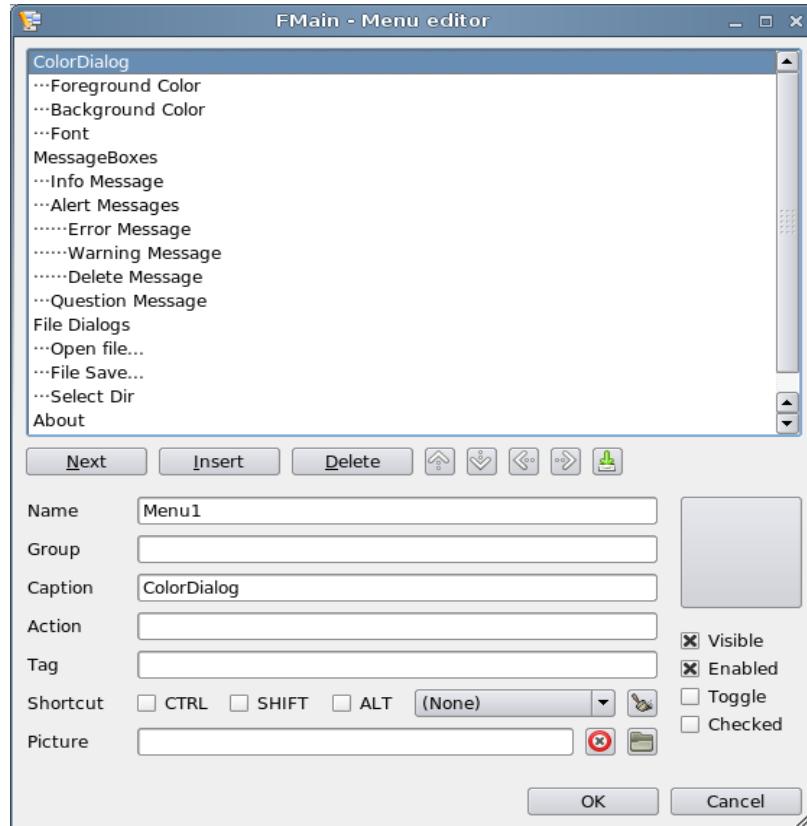


Figure 45: The Gambas Menu Editor.

To begin, click the **Insert** button. First, we will insert the top-level menus, ColorDialog, MessageBoxes, FileDialogs, and Help. To make this easy, use the tables below to fill in the input field data:

Main Menu	(Variable) Name	Group	Caption
ColorDialog	Menu1		ColorDialog
MessageBoxes	Menu2		MessageBoxes
FileDialogs	Menu3		FileDialogs
Help	Menu4		Help

Now, let's put the menu items into the first menu. These menu items will belong to the ColorDialog menu:

ColorDialog	(Variable) Name	Group	Caption
Foreground Color	Menu1Item1		Foreground Color
Background Color	Menu1Item2		Background Color
Font	Menu1Item3		Font

A Beginner's Guide to Gambas – Revised Edition

After you have inserted these items with the editor, ensure they are positioned below the ColorDialog entry and indented one level.

Now, move the menu cursor over the second main menu entry, MessageBoxes and we will insert the menu items for this menu.

MessageBoxes	(Variable) Name	Group	Caption
Info Message	Menu2Item1		Info Message
Alert Messages	Menu2Item2		Alert Messages
Error Message	subMenuItem1		Error Message
Warning Message	subMenuItem2		Warning Message
Delete Message	subMenuItem3		Delete Message
Question Message	Menu2Item3		Question Message

Ensure the Error, Warning, and Delete entries are indented **TWO** levels below the menu entry for Alert Messages.

The following menu items belong to the FileDialogs menu.

FileDialogs	(Variable) Name	Group	Caption
Open File...	Menu3Item1		Open File...
Save File...	Menu3Item2		Save File...
Select Dir	Menu3Item3		Select Dir

Now, the last menu item for the Help menu needs to be added.

Help	(Variable) Name	Group	Caption
About	Menu4		About

That's all there is to it! Click the OK button to leave the menu editor and when you return to the form, a menu will be there. If it does not look like our example, go back to the menu editor and adjust the entries until it does. After you are completely satisfied with the menu's format and appearance, the next step is to code actions for each menu click event. However, before we do that, we need to learn about the standard dialogs and MessageBoxes that are provided in Gambas.

Dialogs

The Dialog class is used to implement all of the Gambas standard dialogs. This class contains static methods used to call the standard dialog boxes. This class is static. The Dialog

methods supported (*i.e., the standard dialogs*) are: OpenFile, SaveFile, SelectColor, SelectDirectory, and SelectFont. The properties supported by this class are: Color, Filter, Font, Path, and Title.

Color is an integer value that the SelectColor Dialog returns representing the color selected. It is interesting to note that Gambas has 32 predefined color constants but the standard SelectColor Dialog consists of 48 colors. This means you cannot use the predefined constants like Red, Black, etc., with any sense of surety in a CASE statement or elsewhere. When a color is chosen, how do you determine if it is a predefined constant or another color outside the range of the 32 predefined in Gambas? You write a module to determine that, of course! We will come back to this topic after explaining how to use the SelectColor Dialog.

When working with the file-related Dialogs, the Filter and Path properties are returned. Filter is defined as **STATIC PROPERTY Filter AS String[]** and it returns or sets the filters (*i.e., categorize by file extension, such as all .doc or .png files*) used in the standard file dialogs. This property returns or receives a string array. Each element in the array represents one filter to use when the file dialog call is made. Each filter string must follow the following syntax:

- ✓ The filter name.
- ✓ A space.
- ✓ An opening bracket.
- ✓ One or more file patterns, separated by semicolons.
- ✓ A closing bracket.

Here is an example that also demonstrates the use of the Dialog Title property:

```
Dialog.Title = "Choose a file"  
Dialog.Filter = ["Pictures (*.png;*.jpg;*.jpeg)","All files (*.*)"]  
Dialog.OpenFile
```

The file-related Dialog calls (SelectDirectory, OpenFileDialog and SaveFileDialog) all return the Path property, which is a string that contains the filepath selected. In the case of the SelectDirectory Dialog call, the string is truncated at the directory level. Now, let's start coding our first menu item, Foreground Color.

From the form window, choose the Foreground Color item from the menu we just built and you will be taken to the code window in a subroutine called **PUBLIC SUB MenuItem1_Click()** where you will want to insert this code:

```
PUBLIC SUB MenuItem1_Click()  
    Dialog.Title = "Choose a foreground color"  
    Dialog.SelectColor  
    TextLabel1.Foreground = Dialog.Color  
    TextLabel1.Text = "<b>Color selected was</b> " & ColorName & "."
```

END

First, we set the Title of the Dialog to “*Choose a foreground color*” so the user has some idea of what the Dialog is used for. We call the standard dialog with **Dialog.SelectColor**.

Once the user has picked a color or canceled from the dialog, we will place the name of the color returned in the **TextLabel1.Text** control we created at the beginning of our project. We choose the **TextLabel** because it allows us to format our output like this:

```
TextLabel1.Text = "<b>Color selected was</b> "&Str$(Dialog.Color) & ". "
```

The line of code above will set to boldface the “**Color selected was**” part of the string, concatenate the Color value to the string after converting the integer to a string using the **Str\$** conversion function, and then concatenate the terminating punctuation to the string (*we do want it to look professional, don't we?*).

Now run the code and you should see something like this:

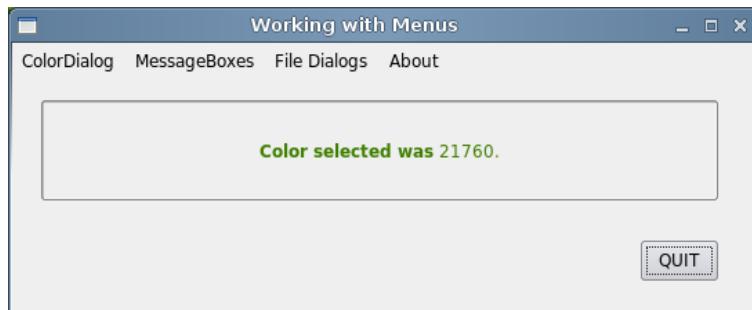


Figure 46: A formatted text label displaying the color value.

Well, 21760 is not very useful information. This number is the internal representation of that greenish color that was selected. We need to develop some method of knowing if the color is red, blue, green, i.e., a named color constant. This is what we were alluding to in the previous section about the problem with colors. Here is where modules come into play.

Modules

We are going to create a module to return the name of a color – if it is a predefined constant. If not, we want a string that tells us it is *not* a predefined color constant. Once we define our module, we will come back and modify our code to use the module. From the Project window, find Modules in the Treeview and select a new module, named Module1. When the dialog appears, leave the CheckBoxes blank and just take the default name and click OK. You will be presented with a new code window titled Module1.module. It should start out with a comment like this:

```
' Gambas module file
```

Now, we will write all the code for our function, which we will call SetColorName. First, we must declare the function and identify what input and output parameters it will process. Here is how we do that:

```
PUBLIC FUNCTION SetColorName ( iVal AS Integer) AS String
```

We declare a public Function that takes an integer parameter and returns a string value. This module will take the integer value that the SelectColor Dialog has returned and check it against the 32 predefined color constants used in Gambas. If it matches a predefined color constant, we will set a temporary string to the name of that color. If not, we will create a string that identifies the constant value and return that to the calling code. We will need to create a temporary string variable to do this, so add this variable declaration as the first line of code inside the function:

```
DIM rval AS String
```

Now, we have to add our CASE SELECT statement. It will use the integer parameter iVal that is passed into the function from the calling subroutine. Here is the full listing for you to enter:

```
' Gambas module file
```

```
PUBLIC FUNCTION SetColorName ( iVal AS Integer) AS String
```

```
'All DIM declarations must be in the FUNCTION or SUB before the first executable command.
```

```
DIM rval AS String
```

```
SELECT iVal
```

```
CASE Color.Black
```

```
    rval = "Black"
```

```
CASE Color.Blue
```

```
    rval = "Blue"
```

```
CASE Color.Cyan
```

```
    rval = "Cyan"
```

```
CASE Color.DarkBlue
```

```
    rval = "DarkBlue"
```

```
CASE Color.DarkCyan
```

```
    rval = "DarkCyan"
```

```
CASE Color.DarkGray
```

```
    rval = "DarkGray"
```

```
CASE Color.DarkGreen
```

```
    rval = "DarkGreen"
```

```
CASE Color.DarkMagenta
```

```
    rval = "DarkMagenta"
```

```
CASE Color.DarkRed
    rval = "DarkRed"
CASE Color.DarkYellow
    rval = "DarkYellow"
CASE Color.Gray
    rval = "Gray"
CASE Color.Green
    rval = "Green"
CASE Color.LightGray
    rval = "LightGray"
CASE Color.Magenta
    rval = "Magenta"
CASE Color.Orange
    rval = "Orange"
CASE Color.Pink
    rval = "Pink"
CASE Color.Red
    rval = "Red"
CASE Color.Transparent
    rval = "Transparent"
CASE Color.Violet
    rval = "Violet"
CASE Color.White
    rval = "White"
CASE Color.Yellow
    rval = "Yellow"
DEFAULT
    rval = Str$(ival) & " and it is not a predefined color constant"
END SELECT
RETURN rval
END
```

Notice that the string variable “**rval**” was dimensioned using the **DIM** keyword, but that the integer variable “**iVal**” was automatically dimensioned when

PUBLIC FUNCTION SetColorName (iVal AS Integer) AS String

was declared. Therefore, “**iVal**” does not use a **DIM** statement.

Once you have completed this code, save this module file and return back to the code window for FMain. We have to modify our code in the **MenuItem1_Click** subroutine. Here is the new subroutine:

```
PUBLIC SUB MenuItem1_Click()  
  
    Dialog.Title = "Choose a foreground color"  
    Dialog.SelectColor  
    ColorName = Module1.SetColorName(Dialog.Color)  
    TextLabel1.ForeColor = Dialog.Color  
    TextLabel1.Text = "<b>Color selected was</b> " & ColorName & ".  
END
```

We have added the line ColorName = ... but have not declared ColorName yet. This variable must be global because we are calling a module to return the value ColorName. At the very beginning of the FMain.class code file, add this code just before the

PUBLIC SUB Form_Open():

```
' Gambas class file  
' declare a global variable to be used with our SetColorName module  
ColorName AS String
```

Now, when the program executes, the variable will be known. In the code below, the call to:

ColorName = Module1.SetColorName(Dialog.Color)

will return the string value assigned to our ColorName variable. We want to set the color of the foreground text to the color just picked:

TextLabel1.ForeColor = Dialog.Color

Now, we will take the string and display it (formatted, of course) by assigning it to the TextLabel1.Text property.

TextLabel1.Text = "Color selected was " & ColorName & ".

The final version of our MenuItem1_Click routine should look like this:

```
PUBLIC SUB MenuItem1_Click()  
    Dialog.Title = "Choose a foreground color"  
    Dialog.SelectColor  
    ColorName = Module1.SetColorName(Dialog.Color)  
    TextLabel1.ForeColor = Dialog.Color  
    TextLabel1.Text = "<b>Color selected was</b> " & ColorName & ".  
END
```

We will want to do the same thing for the background color. The second menu item in

the ColorDialog menu is Background Color so let's click on that from the IDE and code the following:

```
PUBLIC SUB MenuItem2_Click()
    Dialog.Title = "Choose a background color"
    Dialog.SelectColor
    ColorName = Module1.SetColorName(Dialog.Color)
    TextLabel1.BackColor = Dialog.Color
    TextLabel1.Text = "<b>Color selected was</b> " & ColorName & "."
END
```

Note: When you open a new project, Gambas checks your system to determine whether you use the Gnome desktop or KDE. By default, if you are using Gnome, Gambas will load the GTK+ graphic library; if you are using KDE, Gambas loads the Qt library. As of Gambas version 2.2.1, the GTK+ SelectFont widget does not work correctly, and there are other reported gaps in GTK+ integration. For this reason, we recommend opening these projects as “Qt”.

The final option in our first menu is to change the Font. The SelectFont Dialog is used to accomplish this. Once again, go to the form menu and choose the Font option in the ColorDialog menu of our project to get the click event set in the code window. Fonts are handled in Gambas with the Font class. This class represents a font used for drawing or displaying text in controls. This class is creatable. In order for you to declare a font variable, use will need to used this format:

```
DIM hFont AS Font
hFont = NEW Font ( [ Font AS String ] )
```

The code above creates a new font object from a font description. This class acts like a read-only array. This code creates a new font object from a font description and returns it:

```
DIM hFont AS Font
hFont = Font [ Font AS String ]
```

The properties that you can use with Font include:

- ✓ Ascent
- ✓ Bold
- ✓ Descent
- ✓ Fixed
- ✓ Italic
- ✓ Name
- ✓ Resolution
- ✓ Size
- ✓ StrikeOut

- ✓ Styles
- ✓ Underline

Here is an example of how to use code to set the Font properties:

```
Form1.Font.Name = "Utopia"  
Form1.Font.Bold = TRUE  
Form1.Font.Italic = TRUE  
Form1.Font.Size = "14"  
Form1.Font.StrikeOut = FALSE  
Form1.Font.Underline = TRUE
```

The Font class has three Methods you can call: Height, ToString, and Width. **Height** is a function that returns the height of the text displayed with the font. It is declared as:

FUNCTION Height (Text AS String) AS Integer

ToString returns the full name of a font as a description string. This string is a concatenation of all the font properties separated by commas. It is declared as:

FUNCTION ToString () AS String

An example call to ToString would be:

```
PRINT Application.Font.ToString()
```

Width is a function just like Height but it returns the width of the text displayed with the font:

FUNCTION Width (Text AS String) AS Integer

Now, let's continue to enter our code for our MenuProject program in the **PUBLIC SUB MenuItem3_Click()** routine. First, we will declare two local variables, *fontdata* and *oldfontdata*. We want to get the new font from the call to the SelectFont Dialog, but we also want to retain the old font data in case we need it. Next, we declare two string variables, *attr* and *sel* (attribute and selection) to be used to set the font attribute string; selection is a work string we will build dynamically, based on the various selections the user could make in the dialog.

```
PUBLIC SUB MenuItem3_Click()  
DIM fontdata AS font  
DIM oldfontdata AS Font  
DIM attr AS String  
DIM sel AS String
```

A Beginner's Guide to Gambas – Revised Edition

At this point, you may be asking yourself if you could not simply use the `ToString` property to do this. Yes, but you would not learn as much. Bear with me and code along. The following line of code simply blanks out our `attr` string:

```
attr = ""
```

Now, we assign the current `TextLabel1.font` property to the `oldfontdata` variable:

```
oldfontdata = TextLabel1.Font
```

Next, let's set the title for the dialog and call the `SelectFont` Dialog with these two lines:

```
Dialog.Title = " Pick a font... "
Dialog.SelectFont
```

When the user selects a font from the `SelectFont` Dialog, we want to assign the font selected to the `fontdata` variable:

```
fontdata = Dialog.Font
```

At this point, we will check to see what attributes the user has chosen and dynamically build an output string. We could have built this IF THEN ELSE statement to check every property supported by the `Font` class, but we are not interested in all of them for this exercise.

```
IF fontdata.Italic THEN
    sel = " Italic"
    attr = attr & sel
ENDIF
IF fontdata.Bold THEN
    sel = " Bold"
    attr = attr & sel
ENDIF
IF fontdata.Strikeout THEN
    sel = " Strikeout"
    attr = attr & sel
ENDIF
IF fontdata.Underline THEN
    sel = " Underline"
    attr = attr & sel
ENDIF
```

Next, we will set the font in the `TextLabel1` control to be what the user selected and display the dynamically created string of font data to the user:

```
TextLabel1.Font = fontdata  
TextLabel1.Text = "Font: "&fontdata.Name& ", "&Str(Round(fontdata.Size)) & attr
```

We will wait five seconds to let the take a look at the result, then give them a choice as to whether or not they want to keep these settings or revert back to the previous settings.

WAIT 5.0

```
SELECT Message.Question("Keep the new font?", "Yes", "No", "Don't know")
```

CASE 1

```
TextLabel1.Text = "This is now the default font."
```

CASE 2

```
TextLabel1.Font = oldfontdata  
TextLabel1.Text = "Reverted to previous font setting."
```

CASE 3

```
TextLabel1.Font = oldfontdata  
TextLabel1.Text = "No change was made to default font."
```

```
END SELECT
```

```
END
```

Note that we have jumped the gun a bit and introduced the use of MessageBoxes without explaining them. That is our next task, but I wanted to give you a preview. Save your project and execute it at this time. Your results should be similar to this, depending on the colors and font you selected:

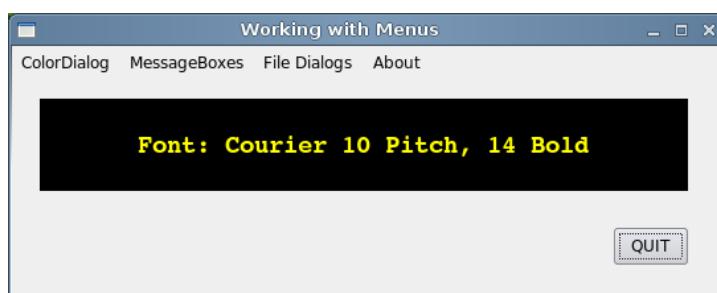


Figure 47: Selecting colors and fonts.

And if you decide to keep it as the default, you will see something like this:



Figure 48: Making a new default font for the `TextLabel1` control.

At this point in your learning journey, you know how to do quite a bit in Gambas. We must continue this journey by expanding our ability to deal with input and output from the user. The MessageBoxes are another valuable tool in our Gambas arsenal. Let's find out more about using them.

MessageBoxes

MessageBox controls are a quite handy means of communicating information to the user or getting answers that are of the TRUE/FALSE, YES/NO type. Generally speaking, MessageBoxes fall into one of four broad categories: query and/or confirmation, information, warning, and error/alert notifications. The **Message class** is used for displaying these MessageBoxes. This class is static and it can be used as a function. It is declared as

STATIC FUNCTION Message (Message AS String [, Button AS String]) AS Integer

The Methods that are supported by the Message class are Info, Question, Delete, Error, and Warning. You have seen the `Message.Question` method used already. Each method serves a specific purpose and should be used appropriately in your code. Let's start with the easiest MessageBox method, the Information method.

Information Messages

The Info method is declared as:

STATIC FUNCTION Info (Message AS String [, Button AS String]) AS Integer

and it is used to display an information MessageBox, with only one button. Since the user does not have to make a decision, they only need to acknowledge the message. Hence, only one button is needed. Here is what our code will produce:

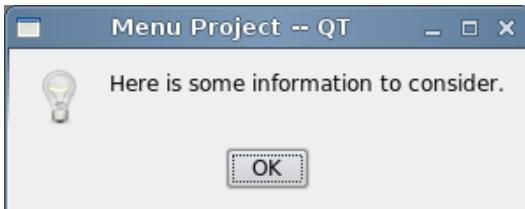


Figure 49: An Information MessageBox.

To continue building our example program, from the form go to the MessageBoxes menu and click on Info Message menu item. It will take you to the code editor window and you will be in the **MenuItem1_Clicked()** subroutine.

```
PUBLIC SUB MenuItem1_Click()
    Message.Info("Here is some information to consider.")
END
```

Pretty simple, eh? What if we wanted to toggle the checked property value from TRUE to FALSE or vice versa each time this menu item was selected. Modify the code above as follows:



Figure 50: A checked menu item.

```
PUBLIC SUB MenuItem1_Click()
IF MenuItem1.Checked THEN
    MenuItem1.Checked = FALSE
ELSE
    MenuItem1.Checked = TRUE
ENDIF
Message.Info("Here is some information to consider.")
END
```

The figure above shows the checked menu item. Selecting it again will uncheck it. Well, that was not too difficult, was it? See, Gambas makes things easy. Now, let's move on to the next part.

Query/Confirm Messages

Question and Delete fall into this category. Question is defined as:

STATIC FUNCTION Question (Message AS String [, Button1 AS String, Button2 AS String, Button3 AS String]) AS Integer

Invoking it displays a question MessageBox with up to three buttons. The index of the button clicked by the user is returned. Let's go back to our previous code and examine it in detail:

```
SELECT Message.Question("Keep the new font?", "Yes","No","Don't know")
CASE 1
    TextLabell.Text = "This is now the default font."
CASE 2
    TextLabell.Font = oldfontdata
    TextLabell.Text = "Reverted to previous font setting."
CASE 3
    TextLabell.Font = oldfontdata
    TextLabell.Text = "No change was made to default font."
END SELECT
```

Because the index of the button clicked by the user is returned, it is easiest to embed the call to Message.Question in a SELECT/CASE statement. SELECT takes the integer return value and, because we know it can only be returned as CASE 1, 2, or 3, we will not need to make a DEFAULT section. We could, of course, but it is not necessary. Depending on the value returned, we will either take a *Yes* action, a *No* action, or a *Don't know* action because that is what we specified in our call to the Dialog (in the SELECT statement above).

Let's write new code for our MessageBoxes Menu menu item Question now. Choose it from the form window and enter this code:

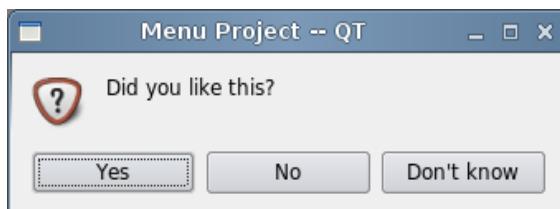


Figure 51: A Question MessageBox.

```
PUBLIC SUB MenuItem3_Click()
SELECT Message.Question("Did you like this?", "Yes","No","Don't know")
CASE 1
    TextLabell.Text = "Liked it."
CASE 2
```

```
TextLabel1.Text = "Did not like it."  
CASE 3  
    TextLabel1.Text = "Did not know."  
END SELECT  
END
```

Notice that in the **SELECT** line, the first text block in quotes is always the MessageBox text, and the next three blocks of text in quotes are always in sequence for Button1, Button2, and Button3. When you execute the program, what you should see is shown above.

Error Messages

The Error function displays an error MessageBox, with up to three buttons. The index of the button clicked by the user is returned. Error is defined as:

STATIC FUNCTION Error (Message AS String[,Btn1 AS String, Btn2 AS String, Btn3 AS String]) AS Integer

Go to the form and click the Error Message subMenuItem to set up a click event and enter this code:

```
PUBLIC SUB subMenuItem1_Click()  
    Message.Error("Wow! This was a mistake.")  
END
```

Here is what you should see when you run the program:



Figure 52: An Error MessageBox.

Warning or Alert Messages

Warning messages display a warning MessageBox, with up to three buttons. The index of the button clicked by the user is returned. Warning is declared as:

STATIC FUNCTION Warning (Message AS String[,Btn1 AS String, Btn2 AS String, Btn3 AS String]) AS Integer

In our example program, error, warning and delete messages are part of the submenu we

created to appear when the menu item Alert Messages is activated.

To code the warning message, we need to set the click event by going to the menu and clicking on the Warning Message subMenuItem. This will put us in the code window where we will enter this code:

```
PUBLIC SUB subMenuItem2_Click()
    Message.Warning("You have been warned about this!")
END
```

When you execute the program, here is what you should see:



Figure 53: A Warning MessageBox.

Delete Messages

The Delete function displays a deletion MessageBox, with up to three buttons. The index of the button clicked by the user is returned. Delete is declared as:

```
STATIC FUNCTION Delete (Message AS String [, Btn1 AS String, Btn2 AS String, Btn3 AS String ]) AS Integer
```

Set the click event for the subMenuItem Delete Message and enter this code:

```
PUBLIC SUB subMenuItem3_Click()
SELECT Message.Delete("Delete?", "Yes","No","Cancel")
CASE 1
    TextLabel1.Text = "Deleted it"
CASE 2
    TextLabel1.Text = "Not Deleted"
CASE 3
    TextLabel1.Text = "Canceled!"
END SELECT
END
```

When you execute the program, here is what you should see:

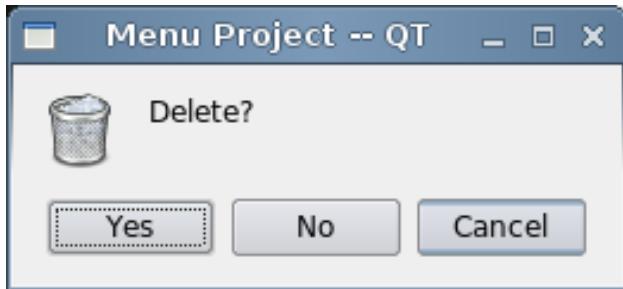


Figure 54: A Delete MessageBox with three buttons.

Dialog Class File-related Functions

The standard dialogs provided with Gambas also support file operation such as opening a file, saving a file, and choosing a directory path. They are pretty standard but the great advantage is that you don't have to build them every time you want to write a program. Consistency in user interface is now demanded from users and anything less makes your program less approachable and, resulting, less accepted by users than it could be. This section will show you how easy it is to use Gambas standard dialogs for file operations.

Dialog OpenFile Function

The OpenFile Dialog function calls the file standard dialog to get the name of a file to open. This function returns TRUE if the user clicked on the Cancel button, and FALSE if the user clicked on the OK button. It is declared as follows:

STATIC FUNCTION OpenFile () AS Boolean

From the example program form window, click on the menu FileDialogs and select the File open... menu item. This sets up the click event and we will enter this code:

```
PUBLIC SUB Menu3Item1_Click()
    Dialog.Title = "Open file..."
    Dialog.OpenFile
    TextLabel1.Text = "OPEN: " & Dialog.Path
END
```

The code above simply sets the title and calls the standard dialog. When the user finishes, the string value returned from the OpenFile Dialog function is assigned to the TextLabel1.Text variable. Here is what it should bring up when you run the code:

A Beginner's Guide to Gambas – Revised Edition

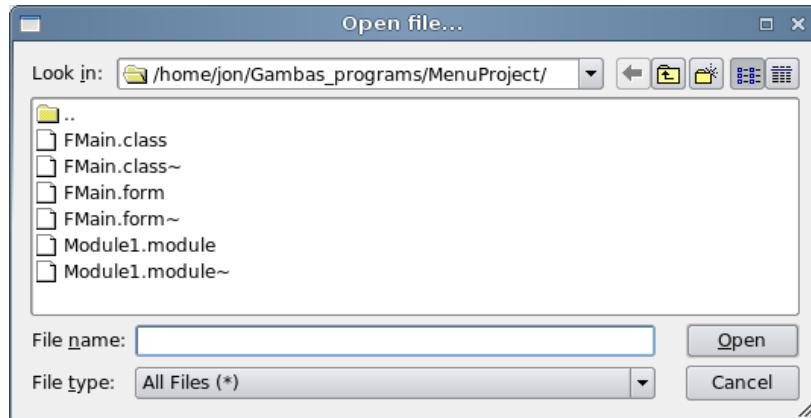


Figure 55: The Open File Dialog in Gambas.

Pretty easy stuff. Saving files is just as easy.

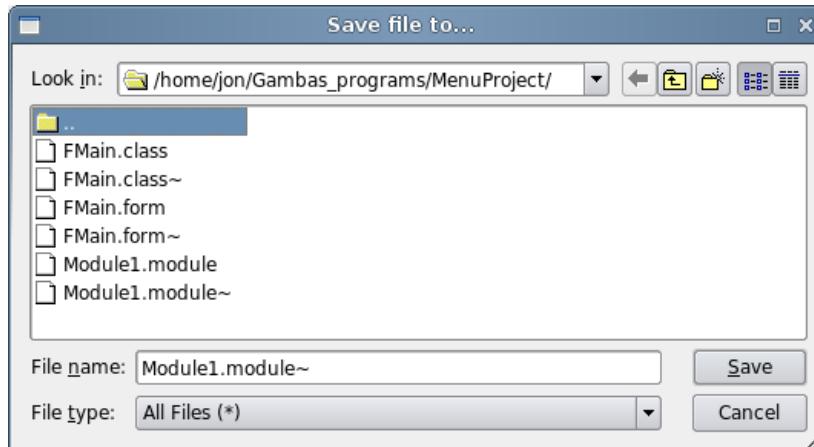


Figure 56: The Save File Dialog in Gambas.

Dialog SaveFile Function

The SaveFile Dialog function also calls the file standard dialog to get the name of a file to save. It returns TRUE if the user clicked on the Cancel button, and FALSE if the user clicked on the OK button. SaveFile is declared as:

STATIC FUNCTION SaveFile () AS Boolean

Create a click event for our File Save... menu item and in the code window enter this code:

```
PUBLIC SUB Menu3Item2_Click()
Dialog.Title = " Save file to... "
Dialog.SaveFile
TextLabel1.Text = "SAVE: " & Dialog.Path
```

END

Dialog SelectDirectory Function

The SelectDirectory Dialog function calls the file standard dialog to get an existing directory name. It returns TRUE if the user clicked on the Cancel button, and FALSE if the user clicked on the OK button. SelectDirectory is declared as:

STATIC FUNCTION SelectDirectory () AS Boolean

This is the final click event we need to code for our program. Choose the Select Dir menu item and enter this code:

```
PUBLIC SUB MenuItem3_Click()
    Dialog.Title = " Pick a dir... "
    Dialog.SelectDirectory
    TextLabel1.Text = "SAVE to DIR: " & Dialog.Path
END
```

Save your project and run the code. Here is what you should see for SelectDir (right).

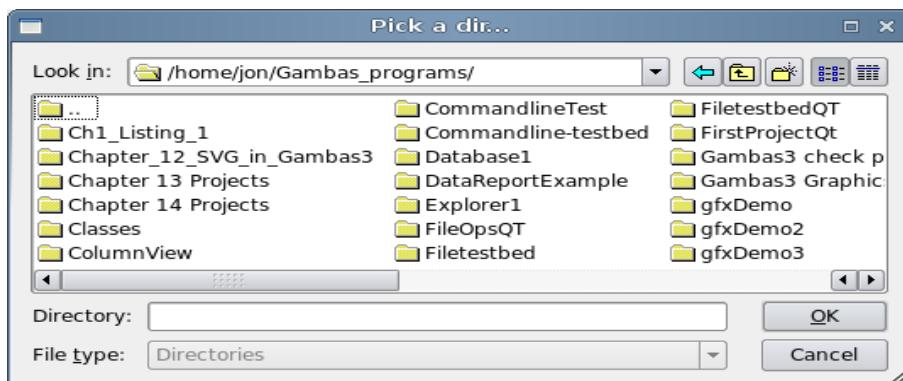


Figure 57: The SelectDirectory dialog.

Now we have written all the code needed to finish our MenuProject program. Save it and execute the program. The next section presents the complete class file for you to review. It is time for us to move on to bigger and better things. In the next chapter, we will start learning how to handle input and output from the user by using strings and files.

Complete Example Listing

```
' Gambas class file
' declare a global variable to be used with our SetColorName module
ColorName AS String      'A global variable does not use the DIM keyword, and it must come before
                           'the PUBLIC SUB Form_Open() statement.

PUBLIC SUB Form_Open()
    ME.Caption = " Working with Menus "
END

PUBLIC SUB QuitBtn_Click()
    FMain.Close
END

PUBLIC SUB MenuItem1_Click()
    Dialog.Title = "Choose a foreground color"
    Dialog.SelectColor
    ColorName = Module1.SetColorName(Dialog.Color)
    TextLabel1.ForeColor = Dialog.Color
    TextLabel1.Text = "<b>Color selected was</b> " & ColorName & "."
END

PUBLIC SUB MenuItem2_Click()
    Dialog.Title = "Choose a background color"
    Dialog.SelectColor
    ColorName = Module1.SetColorName(Dialog.Color)
    TextLabel1.BackColor = Dialog.Color
    TextLabel1.Text = "<b>Color selected was</b> " & ColorName & "."
END

PUBLIC SUB MenuItem3_Click()
    DIM fontdata AS font
    DIM oldfontdata AS Font
    DIM attr AS String
    DIM sel AS String
    attr = ""
    oldfontdata = TextLabel1.Font

    Dialog.Title = " Pick a font... "
    Dialog.SelectFont
    fontdata = Dialog.Font
    IF fontdata.Italic THEN
```

```

sel = " Italic"
attr = attr & sel
ENDIF
IF fontdata.Bold THEN
    sel = " Bold"
    attr = attr & sel
ENDIF
IF fontdata.Strikeout THEN
    sel = " Strikeout"
    attr = attr & sel
ENDIF
IF fontdata.Underline THEN
    sel = " Underline"
    attr = attr & sel
ENDIF
TextLabel1.Font = fontdata
TextLabel1.Text = "Font: " & fontdata.Name & ", " & Str(Round(fontdata.Size)) & attr
WAIT 5.0
SELECT Message.Question("Keep new font?", "Yes","No","Don't know")
CASE 1
    TextLabel1.Text = "This is now the default font."
CASE 2
    TextLabel1.Font = oldfontdata
    TextLabel1.Text = "Reverted to previous font setting."
CASE 3
    TextLabel1.Font = oldfontdata
    TextLabel1.Text = "No change was made to default font."
END SELECT
END

PUBLIC SUB MenuItem1_Click()
IF MenuItem1.Checked THEN
    MenuItem1.Checked = FALSE
ELSE
    MenuItem1.Checked = TRUE
ENDIF
Message.Info("Here is some information to consider.")
END

PUBLIC SUB subMenuItem1_Click()
Message.Error("Wow! This was a mistake.")
END

```

A Beginner's Guide to Gambas – Revised Edition

```
PUBLIC SUB subMenuItem2_Click()
    Message.Warning("You have been warned about this!")
END

PUBLIC SUB subMenuItem3_Click()
    SELECT Message.Delete("Delete?", "Yes", "No", "Cancel")
    CASE 1
        TextLabell.Text = "Deleted it"
    CASE 2
        TextLabell.Text = "Not Deleted"
    CASE 3
        TextLabell.Text = "Cancelled!"
    END SELECT
END

PUBLIC SUB Menu2Item3_Click()
    SELECT Message.Question("Did you like this?", "Yes", "No", "Don't know")
    CASE 1
        TextLabell.Text = "Liked it."
    CASE 2
        TextLabell.Text = "Did not like it."
    CASE 3
        TextLabell.Text = "Did not know."
    END SELECT
END

PUBLIC SUB Menu3Item1_Click()
    Dialog.Title = " Open file... "
    Dialog.OpenFile
    TextLabell.Text = " OPEN: " & Dialog.Path
END

PUBLIC SUB Menu3Item2_Click()
    Dialog.Title = " Save file to... "
    Dialog.SaveFile
    TextLabell.Text = "SAVE: " & Dialog.Path
END

PUBLIC SUB Menu3Item3_Click()
    Dialog.Title = " Pick a dir... "
    Dialog.SelectDirectory
    TextLabell.Text = "SAVE to DIR: " & Dialog.Path
END
```

```
PUBLIC SUB Menu4Item1_Click()
    Message.Info("This is all about<br><b> Gambas Programming</b>! ")
END
```

MMain.module listing

```
' Gambas module file
PUBLIC FUNCTION SetColorName ( iVal AS Integer ) AS String
    DIM rval AS String      'A local variable must be dimensioned in the FUNCTION or SUB before the first executable
command.
    SELECT iVal
    CASE Color.Black
        rval = "Black"
    CASE Color.Blue
        rval = "Blue"
    CASE Color.Cyan
        rval = "Cyan"
    CASE Color.DarkBlue
        rval = "DarkBlue"
    CASE Color.DarkCyan
        rval = "DarkCyan"
    CASE Color.DarkGray
        rval = "DarkGray"
    CASE Color.DarkGreen
        rval = "DarkGreen"
    CASE Color.DarkMagenta
        rval = "DarkMagenta"
    CASE Color.DarkRed
        rval = "DarkRed"
    CASE Color.DarkYellow
        rval = "DarkYellow"
    CASE Color.Gray
        rval = "Gray"
    CASE Color.Green
        rval = "Green"
    CASE Color.LightGray
        rval = "LightGray"
    CASE Color.Magenta
        rval = "Magenta"
    CASE Color.Orange
        rval = "Orange"
```

```
CASE Color.Pink
    rval = "Pink"
CASE Color.Red
    rval = "Red"
CASE Color.Transparent
    rval = "Transparent"
CASE Color.Violet
    rval = "Violet"
CASE Color.White
    rval = "White"
CASE Color.Yellow
    rval = "Yellow"
DEFAULT
    rval = Str$(ival) & " and it is not a predefined color constant"
END SELECT
RETURN rval
END
```

In the next chapter, we will learn how to handle strings and perform various data conversion activities that are common in programming. The advantage you have in dealing with strings, however, is that you are now programming using Gambas. It makes the task much easier than many conventional programming languages, as you will soon see.



Chapter 7 – Strings and Data Conversion

One of the most important things a programmer needs to know is how to handle strings and use the built-in functions provided by the development language he or she may be working with. Manipulating strings and converting from one data-type to another is almost always required when programming. The more adept a programmer is at using the built-in functions, the more likely it is their program will operate efficiently.

String Functions

In Gambas, there is a rich set of string functions. We will use the console for this chapter and learn about the following functions:

- ✓ Len
- ✓ Upper\$/Ucase\$ and Lower\$/LCase\$
- ✓ Trim\$/RTrim\$ and LTrim\$
- ✓ Left\$/Mid\$/Right\$
- ✓ Space\$
- ✓ Replace\$
- ✓ String\$
- ✓ Subst\$
- ✓ InStr
- ✓ RInStr
- ✓ Split

We will need to create a new console application to do our work in this chapter. Start Gambas and create a new terminal application named StringTests. When the IDE appears, double-click on the MMain document icon to create a new startup class named MMain.module. The code window should appear and you should see something like this:

```
' Gambas module file
PUBLIC SUB Main()
END
```

We now have all the preliminaries out of the way and are ready to begin learning about the strings in Gambas. You may remember from an earlier chapter that in Gambas a string data-type is a reference to a variable length string. It is four bytes in initial size and when it is created it is initialized with a Null value.

Len

Len returns the length of a string. The return value is an Integer. Here is how **Len**

would be used in code:

Length = Len (String)

From the code window, let's type the following:

```
PUBLIC SUB Main()
DIM iStringLength AS Integer
DIM sTestString AS String

sTestString = "12345678901234567890"
iStringLength = Len(sTestString)
PRINT "==> " & iStringLength & " is the length of our test string."

sTestString = "12345"
iStringLength = Len(sTestString)
PRINT "==> " & iStringLength & " is the length of our test string."

sTestString = "12345678901"
iStringLength = Len(sTestString)
PRINT "==> " & iStringLength & " is the length of our test string."
END
```

The console will respond with this:

```
==> 20 is the length of our test string.
==> 5 is the length of our test string.
==> 11 is the length of our test string.
```

It is important to know that string length begins counting at position 1, not zero like in C, Python and some other languages. Now that we can find out how many characters are in a string, let's learn how to convert the string from one case to another.

Upper\$/Ucase\$/Ucase - Lower\$/Lcase\$/Lcase

Upper\$ returns a string converted to upper case. Ucase\$ and Ucase are synonyms for Upper\$ and can be used interchangeably. Lower\$ returns a string converted to lower case. Lcase\$ and Lcase are synonyms for Lower\$ and can also be used interchangeably. Note that these functions **do not work** with UTF-8 strings with a character code higher than 128. Here is the standard Gambas language syntax for using these functions:

Result = Upper\$ (String)

Result = UCase\$ (String)

A Beginner's Guide to Gambas – Revised Edition

Type this in the code window and execute the program:

```
PUBLIC SUB Main()  
  
DIM sTestString AS String  
  
sTestString = "abcdefg"  
  
PRINT "==> " & sTestString & " is our starting string."  
PRINT "==> " & UCASE$(sTestString) & " is now uppercase."  
PRINT "==> " & LCASE$(sTestString) & " is now back to lowercase."  
PRINT "==> " & UPPER$(sTestString) & " is now back to uppercase."  
  
sTestString = "123abc456def 789ghiZZZ"  
  
PRINT "==> " & sTestString & " is our starting string."  
PRINT "==> " & UCASE(sTestString) & " is now uppercase."  
PRINT "==> " & LCASE(sTestString) & " is now back to lowercase."  
PRINT "==> " & UPPER(sTestString) & " is now back to uppercase."  
PRINT "==> " & LOWER$(sTestString) & " is now back to lowercase."  
END
```

The console responds with:

```
==> abcdefg is our starting string.  
==> ABCDEFG is now uppercase.  
==> abcdefg is now back to lowercase.  
==> ABCDEFG is now back to uppercase.  
==> 123abc456def 789ghiZZZ is our starting string.  
==> 123ABC456DEF 789GHIZZZ is now uppercase.  
==> 123abc456def 789ghizzz is now back to lowercase.  
==> 123ABC456DEF 789GHIZZZ is now back to uppercase.  
==> 123abc456def 789ghizzz is now back to lowercase.
```

Trim\$, LTrim\$, and RTrim\$

Trim\$ strips away all white spaces from either end of a string. It will not strip white spaces away after the first non-whitespace is encountered and basically ignores all whitespace until the last non-whitespace character is encountered, whereupon it will begin to trim trailing whitespace. A white space is any character whose ASCII code is strictly lower than 32. Trim\$ is basically a call to **Ltrim\$** and **Rtrim\$** simultaneously. It is used like this:

```
Result = Trim$(String)
```

Try this on your console see how the Trim\$ function works:

```
PUBLIC SUB Main()
    DIM sTestString AS String
    DIM sResult AS String
    DIM iLength AS Integer

    PRINT "      1      2"
    PRINT "12345678901234567890"
    sTestString = " <abcdef "
    iLength = Len(sTestString)

    PRINT sTestString & " is our starting string."
    PRINT "It is " & Str$(iLength) & " characters long"
    sResult = Trim$(sTestString)
    PRINT sResult & "> is the result of Trim$ call."
END
```

The console responds with this:

```
      1      2
12345678901234567890
<abcdef> > is our starting string.
It is 13 characters long
<abcdef> is the result of Trim$ call.
```

Left\$, Mid\$, and Right\$

Left\$ returns the first *Length* characters of a string. If *Length* is not specified, the first character of the string is returned. If *Length* is negative, all of the string except the *-Length* last characters are returned. Standard Gambas language syntax is:

```
Result = Left$ ( String [ , Length ] )
```

Mid\$ returns a substring containing the *Length* characters from the first position. If *Length* is not specified, everything from the start position is returned. If *Length* is negative, everything from the start position except the *-Length* last characters is returned. Standard Gambas language syntax is:

```
Result = Mid$ ( String , Start [ , Length ] )
```

Right\$ returns the *Length* last characters of a string. If *Length* is not specified, the last character of the string is returned. If *Length* is negative, all of the string except the

-*Length* first characters are returned. Standard Gambas language syntax is:

Result = Right\$ (String [, Length])

Try this on your console:

```
PUBLIC SUB Main()
DIM sTestString AS String
DIM sResult AS String
DIM iLength AS Integer

PRINT "      1      2"
PRINT "12345678901234567890"
sTestString = "abcdefghijklm"
iLength = Len(sTestString)

PRINT sTestString & " is our starting string."
PRINT "It is " & Str$(iLength) & " characters long"

sResult = Left$(sTestString,3)
PRINT sResult & " is the result of Left$ call."
sResult = Mid$(sTestString,4,3)
PRINT sResult & " is the result of Mid$ call."
sResult = Right(sTestString,3)
PRINT sResult & " is the result of Right$ call."
END
```

The console responds with this;

```
      1      2
12345678901234567890
abcdefghijklm is our starting string.
It is 9 characters long
abc is the result of Left$ call.
def is the result of Mid$ call.
ghi is the result of Right$ call.
```

Space\$

Space\$ returns a string containing *Length* spaces. Standard Gambas language syntax is:

String = Space\$ (Length)

Try this on the console:

```
PUBLIC SUB Main()
DIM sTestString AS String
DIM sResult AS String

PRINT "      1      2"
PRINT "12345678901234567890123456"
sTestString = "a"
PRINT sTestString & Space$(24) & "z"
END
```

The console responds with:

```
      1      2
12345678901234567890123456
a                  z
```

Replace\$

Replace\$ replaces every occurrence of the string Pattern in the string String by the string ReplaceString , and returns the result. If String is null, then a null string is returned. If Pattern is null, then the string String is returned. Standard Gambas language syntax is as follows:

```
Result = Replace$ ( String , Pattern , ReplaceString )
```

Try this program on your console:

```
PUBLIC SUB Main()
DIM sTestString AS String
DIM sResult AS String

sTestString = "abc123ghi"
PRINT sTestString & " is our starting string."
sResult = Replace$(sTestString,"123","def")
PRINT sResult & " is the result of Replace call."

sTestString = "a b c d e f g h i"
PRINT sTestString & " is our starting string."
sResult = Replace$(sTestString," ","-")
PRINT sResult & " is the result of Replace call."
```

```
sTestString = "\ta\tb\tc\tdef"
PRINT sTestString & " is our starting string."
sResult = Replace$(sTestString, "\t", "")
PRINT sResult & " is the result of Replace call."
END
```

The console responds with:

```
abc123ghi is our starting string.
abcdefghi is the result of Replace call.
a b c d e f g h i is our starting string.
a-b-c-d-e-f-g-h-i is the result of Replace call.
      A      b      c      def is our starting string.
abcdef is the result of Replace call.
```

String\$

String\$ simply returns a string containing Length times the Pattern. Use this format:

```
String = String$ ( Length , Pattern )
```

Try this on the console:

```
PUBLIC SUB Main()
DIM sTestString AS String
DIM sResult AS String

PRINT "    1    2"
PRINT "12345678901234567890123456"
sTestString = "a"

PRINT sTestString & String(24,".") & "z"
END
```

The console responds with:

```
    1    2
12345678901234567890123456
a.....z
```

Subst\$

Subst\$ replaces arguments &1, &2, etc. in a pattern with the first, second, and subsequent ReplaceStrings respectively, and return the result. If Pattern is null, then a null string is returned. For C developers, this is not unlike a simplified sprintf. This function is

A Beginner's Guide to Gambas – Revised Edition

very useful when you must concatenate strings that must be translated. Do not use the & operator inside the parentheses, containing the Pattern and ReplaceStrings. When you assign strings to an order in the Subst statement, be careful with the order of words and phrases, as they may be reversed in different languages. Standard Gambas language syntax is:

Result = Subst\$(Pattern , ReplaceString [, ReplaceString])

Try this little application on your console:

```
PUBLIC SUB Main()

DIM LangF, IamF, LangG, IamG, LangE, IamE AS String

LangF = "en Français:" 'in French
IamF = "Je suis programmeur du Gambas"
LangG = "auf Deutsch:" 'in German
IamG = "Ich bin Gambas Programmierer"
LangE = "in English:"
IamE = "I am a Gambas programmer"

PRINT Subst$("French -- &1 &2", LangF, IamF) & "!"
PRINT Subst$("German -- &1 &2", LangG, IamG) & "!"
PRINT Subst$("English -- &2 &1", IamE, LangE) & "!"

END
```

The console responds with:

```
French -- en Français: Je suis programmeur du Gambas!
German -- auf Deutsch: Ich bin Gambas Programmierer!
English -- in English: I am a Gambas programmer!
```

If you are dimensioning multiple variables of the same data-types, you don't have to put them all in separate lines. You can put several on one line, separated by commas, with the type declaration at the end:

```
DIM a, b, c,...x AS String
```

Observe that in the English line in the code, the Patterns and the ReplaceString variables are intentionally reversed, but the output in the console is correct. There is no required order of the ReplaceString variables in the Subst\$ statement. The numbers in the Pattern are tags, which refer to the variables in the ordinal position where you decided to put them. If it makes sense to order the Pattern/ ReplaceString as "&2 &1", or "&3 &1 &2", that is also acceptable.

InStr

InStr returns the position of the first occurrence of Substring in String. If Start is specified, the search begins at the position Start. If the substring is not found, InStr() returns zero.

Position = InStr (String , Substring [, Start])

The following code is a bit more involved and will demonstrate the power of this function. We want to find every space (set in the string at even-numbered positions) and print out the position where each space occurs. Enter this code in the console code window:

```
' Gambas module file
PUBLIC SUB Main()
DIM sTestString AS String
DIM sResult AS String
DIM iLength AS Integer
DIM iPosition AS Integer
DIM iNextCharPos AS Integer
DIM iCounter AS Integer

sTestString = "abc def ghi"
iPosition = Instr(sTestString, " ")
PRINT sTestString & " is our start string."
PRINT "First space is at position: " & iPosition
iNextCharPos = iPosition + 1
iPosition = Instr(sTestString, " ",iNextCharPos)
PRINT "Next space is at position: " & iPosition
PRINT

sTestString = "a b c d e f g h i j k l m n o p q r s t u v w x y z"
PRINT "      1      2      3      4      5      6"
PRINT "123456789012345678901234567890123456789012345678901234567890"
PRINT sTestString & " is our new test string."
PRINT
iLength = Len(sTestString)
PRINT "Length of sTestString is: " & iLength

iPosition = Instr(sTestString, " ")
PRINT "First space is at position: " & iPosition
FOR iCounter = iPosition TO iLength/2
    iNextCharPos = iPosition + 1
    iPosition = Instr(sTestString, " ",iNextCharPos)
```

A Beginner's Guide to Gambas – Revised Edition

```
PRINT "Next space is at position: " & iPosition  
NEXT  
END
```

The resulting output is:

```
abc def ghi is our start string.  
First space is at position: 4  
Next space is at position: 8  
  
1 2 3 4 5 6  
12345678901234567890123456789012345678901234567890  
a b c d e f g h i j k l m n o p q r s t u v w x y z is our new test string.
```

```
Length of sTestString is: 51  
First space is at position: 2  
Next space is at position: 4  
Next space is at position: 6  
. .  
Next space is at position: 46  
Next space is at position: 48  
Next space is at position: 50
```

RInStr

RInStr returns the position of the last occurrence of Substring in String, searching from right to left. If Start is specified, the search stops at the position Start. If the substring is not found, RInStr() returns zero. Standard Gambas language syntax is:

```
Position = RInStr ( String , Substring [ , Start ] )
```

Enter this code in the console:

```
PUBLIC SUB Main()  
DIM sTestString AS String  
  
DIM iPosition AS Integer  
  
sTestString = "abc def abc def abc"  
PRINT "1 2"  
PRINT "12345678901234567890"  
PRINT sTestString
```

```
iPosition = RInstr(sTestString,"abc")
PRINT
PRINT "last occurrence of abc starts at position: " & iPosition
END
```

The console responds with:

```
1      2
12345678901234567890
abc def abc def abc

last occurrence of abc starts at position: 17
```

Split

Split splits a string into substrings delimited by the separator(s) designated as parameters. Escape characters can be specified also. Any separator characters enclosed between two escape characters are ignored in the splitting process. Note that Split takes only three arguments so if you want to use several separators, you should pass them as the second parameter, concatenated in a single string. By default, the comma character is the separator, and there are no escape characters. This function returns a string array filled with each detected substring. The Gambas language syntax is:

```
Array = Split ( String [, Separators , Escape ] )
```

Here is a program to try on your console:

```
PUBLIC SUB Main()
DIM aWordArray AS String[]
DIM sWord AS String

' note we use a space delimiter
aWordArray = Split("This is indeed a very cool feature of Gambas!"," ")

FOR EACH sWord IN aWordArray
    PRINT sWord
NEXT
END
```

The console responds with:

```
This
is
```

```
indeed  
a  
very  
cool  
feature  
of  
Gambas!
```

Data Conversion

Asc and Chr\$

Asc returns the ASCII code of the character at position *Position* in the String . If *Position* is not specified, the ASCII code of the first character is returned. Chr\$ returns the character whose ASCII code is *Code*. A word of caution: Gambas uses the UTF-8 charset internally, so any character code greater than 128 may not have the same meaning as they would have with another charset (for example, ISO8859-1). Here is the Gambas language syntax:

```
Code = Asc ( String [ , Position ] )  
Character = Chr$ ( Code )
```

Here is a sample program that illustrates the use of both ASC and CHR\$:

```
' Gambas module file  
PUBLIC SUB Main()  
DIM iAsciiCode AS Integer  
DIM sTestString AS String  
DIM iLength AS Integer  
DIM counter AS Integer  
  
sTestString = "Gambas is great."  
iLength = Len(sTestString)  
PRINT "Length of sTestString is: " & Str$(iLength)  
FOR counter = 1 TO iLength  
    iAsciiCode = Asc(sTestString,counter)  
    IF iAsciiCode <> 32 THEN  
        PRINT iAsciiCode & " is char: " & Chr(9) & Chr$(iAsciiCode)  
    ELSE  
        PRINT iAsciiCode & " is char: " & Chr(9) & "<space>"  
    ENDIF  
NEXT  
END
```

In the code above, Chr(9) is used to represent the TAB character. The resulting output from the console is:

```
Length of sTestString is: 16
71 is char:   G
97 is char:   a
109 is char:  m
98 is char:   b
97 is char:   a
115 is char:  s
32 is char:  <space>
105 is char: i
115 is char: s
32 is char:  <space>
103 is char: g
114 is char: r
101 is char: e
97 is char:   a
116 is char: t
46 is char:  .
```

Bin\$

Bin\$ returns the binary representation of a number. If Digits is specified, the representation is padded with unnecessary zeros so that Digits digits are returned. Here is the calling convention for Bin\$:

String = Bin\$ (Number [, Digits])

Example code you can try on the console:

```
PUBLIC SUB Main()
  DIM sBinaryCode AS String
  DIM counter AS Integer

  FOR counter = 0 TO 15
    sBinaryCode = Bin$(counter,4)
    PRINT sBinaryCode
  NEXT
END
```

The output should look like this:

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

CBool

CBool converts an expression into a Boolean value. An expression is false if any of the following conditions are met:

- ✓ A false Boolean value
- ✓ A zero number
- ✓ A zero length string
- ✓ A null object

Otherwise, the expression is true in all other cases. Here is the Gambas language syntax:

Boolean = CBool (Expression)

Here is an example:

```
PUBLIC SUB Main()
    PRINT CBool(0); " "; CBool(1)
END
```

The console responds with:

FALSE TRUE

CByte

CByte converts an expression into a byte. *Expression* is first converted into an integer.

Then, if this integer overflows the byte range, (-32767 to 32768) it is truncated.

Byte = CByte (Expression)

Example:

```
PUBLIC SUB Main()
PRINT CByte("17")
PRINT CByte(32769)
PRINT CByte(TRUE)
END
```

The console responds with:

```
17
1
255
```

CDate

CDate converts an expression into a date/time value. Be careful! The current localization is NOT used by this function. Gambas language syntax is:

Date = CDate (Expression)

Example:

```
PUBLIC SUB Main()
DIM sDateString AS String
sDateString = CDate(Now)
PRINT sDateString; " is our start time."
WAIT 1.0
PRINT CDate(Now); " is one second later."
PRINT CDate(sDateString); " is still where we started."
WAIT 1.0
PRINT Now; " is one second later."
END
```

The console responds with:

```
08/21/2005 20:52:40 is our start time.
08/21/2005 20:52:41 is one second later.
08/21/2005 20:52:40 is still where we started.
```

08/21/2005 20:52:42 is one second later.

CFloat

CFloat converts an expression into a floating point number. Be careful! The current localization is NOT used by this function. Gambas language syntax is:

Float = CFloat (Expression)

Example:

```
PUBLIC SUB Main()
  DIM sFloatString AS String
  DIM fFloatNum AS Float

  sFloatString = "0.99"
  fFloatNum = 0.01

  PRINT fFloatNum + CFloat(sFloatString)
  PRINT CFloat("3.0E+3")
END
```

The console responds with:

```
1
3000
```

Considering localization (L10N) issues for the use of CFLOAT, the different usage of “,” and “.” between the U.S. and European languages relies on the default code page that is in use when the application is executed. The code works but is entirely dependent on the native code page used on the executable platform, not the design/programmer platform. When programming, should L10N be a consideration, always consider what the end-user platform should be and program accordingly. We will discuss internationalization (I18N) and L10N later in this book.

CInt / Cinteger and CShort

CInt is synonymous with CInteger. Either call converts an expression into an integer. The standard Gambas language syntax is:

Integer = CInt (Expression)
Integer = CInteger (Expression)

CShort converts an expression into a short integer. Expression is first converted into an

integer. Then, if this integer overflows the short range, it is truncated. Here is the Gambas language syntax for CShort:

Short = CShort (Expression)

Here is an example that slightly modifies the previous console example to show both of these functions:

```
PUBLIC SUB Main()
DIM sFloatString AS String
DIM fFloatNum AS Float

sFloatString = "0.99"
fFloatNum = 120.901

PRINT fFloatNum + CFloat(sFloatString)
PRINT CInt(fFloatNum); " was a float, now an int."
PRINT CShort(fFloatNum); " was a float, now a short."
END
```

The result is:

```
121.891
120 was a float, now an int.
120 was a float, now a short.
```

CStr / CString

CStr / CString converts an expression into a string. Be careful! The current localization is NOT used by this function. Here is how the Gambas language syntax for CStr works:

```
String = CStr ( Expression )
String = CString ( Expression )
```

Example:

```
PUBLIC SUB Main()
DIM sFloatString AS String
DIM fFloatNum AS Float

fFloatNum = 120.901
sFloatString = CStr(fFloatNum)
PRINT sFloatString; " is now a string."
END
```

Console output is:

120.901 is now a string.

Hex\$

Hex\$ returns the hexadecimal representation of a number. If *Digits* is specified, the representation is padded with unnecessary zeros so that *Digits* digits are returned. Here is the Gambas language syntax:

String = Hex\$(Number [, Digits])

Here is an example program:

```
PUBLIC SUB Main()
DIM counter AS Integer

FOR counter = 1 TO 15
    PRINT Hex$(counter,2);";"
NEXT
END
```

This is the result:

01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

Conv\$

Conv\$ converts a string from one charset to another. A charset is represented by a string like "ASCII", "ISO-8859-1", or "UTF-8". The GAMBAS interpreter internally uses the UTF-8 charset. The charset used by the system is returned by a call to System.Charset. It is ISO-8859-1 on Mandrake 9.2 operating system, but it is UTF-8 on a RedHat Linux operating system. In the future, nearly all Linux systems will probably be UTF-8 based. The charset used by the graphical user interface is returned by a call to Desktop.Charset. It is UTF-8 with the Qt component. The Gambas language syntax is:

ConvertedString = Conv\$(String AS String, SourceCharset AS String, DestinationCharset AS String)

The conversion relies on the **iconv()** GNU library function. Here is a code example to try:

```
PUBLIC SUB Main()
DIM sStr AS String
```

```
DIM iInd AS Integer  
  
sStr = Conv$("Gambas", "ASCII", "EBCDIC-US")  
FOR iInd = 1 TO Len(sStr)  
    PRINT Hex$(Asc(Mid$(sStr, iInd, 1)), 2); " ";  
NEXT  
END
```

Here is the output:

C7 81 94 82 81 A2

Val and Str\$

Val converts a string into a Boolean, a number or a date, according to the content of the string. The current localization is used to convert numbers and dates. The conversion algorithm uses the following order of precedence:

```
If string can be interpreted as a date & time (with date or time separators), then the date & time is returned.  
Else if string can be interpreted as a floating point number, then a floating point number is returned.  
Else if string can be interpreted as a integer number, then an integer number is returned.  
Else if string is TRUE or FALSE, the matching Boolean value is returned.  
Else NULL is returned.
```

Standard Gambas language syntax is:

Expression = Val (String)

Str\$ converts an expression into its printable string representation. It is the exact contrary of Val(). The current localization is used to convert numbers and dates. Gambas language syntax is:

String = Str\$ (Expression)

Try this:

```
PUBLIC SUB Main()  
DIM sDateTime AS String  
DIM dDate AS Date  
  
PRINT Now; " is current system time."  
  
sDateTime = Val(Str$(Now))  
PRINT sDateTime; " is a string representation of current system time."  
PRINT Val(sDateTime); " is VAL conv of the string representation."
```

```
dDate = Val(sDateTime)
PRINT dDate; " is the DATE variable converted with VAL."
PRINT Str(dDate); " is a string representation of the date variable."
END
```

Here is the output:

```
08/21/2005 21:42:45 is current system time.
08/21/2005 21:42:45 is a string representation of current system time.
08/21/2005 21:42:45 is VAL conv of the string representation.
08/21/2005 21:42:45 is the DATE variable converted with VAL.
08/21/2005 21:42:45 is a string representation of the date variable.
```

Here is another sample program to try:

```
' Gambas module file

PUBLIC SUB Main()

DIM sInputLine AS String
DIM value AS Variant

DO WHILE sInputLine <> "quit"
    PRINT "==> ";
    LINE INPUT sInputLine
    IF sInputLine = "" THEN
        sInputLine = "<CRLF>"
    ENDIF
    PRINT "You typed: "; sInputLine
    value = Val(sInputLine)
    PRINT
    IF IsBoolean(value) THEN
        PRINT sInputLine; " is Boolean."
    ELSE IF IsDate(value) THEN
        PRINT sInputLine; " is a date."
    ELSE IF IsInteger(value) THEN
        PRINT sInputLine; " is an Integer."
        IF value = 0 OR value = 1 THEN
            PRINT "It could also be Boolean."
        ENDIF
        IF value > 0 AND value < 255 THEN
            PRINT "It could also be a Byte."
        ENDIF
    ENDIF
END
```

```
ENDIF
IF value > -32767 AND value < 32768 THEN
    PRINT "It could also be a short."
ENDIF
PRINT "-----"
ELSE IF IsFloat(value) THEN
    PRINT sInputLine; " is a float."
ELSE IF IsString(value) THEN
    PRINT sInputLine; " is a string."
ELSE IF IsNull(value) THEN
    PRINT sInputLine; " is NULL."
ELSE
    PRINT sInputLine; " is something else."
ENDIF
LOOP
PRINT
PRINT "We are done!"
END
```

The output should be similar to this:

```
==> true
You typed: true

true is Boolean.
==> 214
You typed: 214

214 is an Integer.
It could also be a Byte.
It could also be a short.
-----
==> 08/23/05 12:23:55
You typed: 08/23/05 12:23:55

08/23/05 12:23:55 is a date.
==> John
You typed: John

John is a string.
==> -32756
You typed: -32756
```

-32756 is an Integer.
It could also be a short.

==> quit
You typed: quit

quit is a string.

We are done!

Note that the IsString function (used above and discussed below) is apparently eliminated in Gambas3.

Format\$

Format\$ converts an expression to a string by using a format that depends on the type of the expression. Format can be a predefined format (an integer constant) or a user-defined format (a string that depicts the format). If Format is not specified, gb.Standard component formats are used. This function uses localization information to format dates, times and numbers. The Gambas language syntax is:

String = Format\$ (Expression [, Format])

A user-defined number format is described by the following characters:

Format Symbol	Meaning
"+"	prints the sign of the number.
"-"	prints the sign of the number only if it is negative.
"#"	prints a digit only if necessary.
"0"	always prints a digit, padding with a zero if necessary.
"."	prints the decimal separator.
"%"	multiples the number by 100 and prints a per-cent sign.
"E"	introduces the exponential part of a float number. The sign of the exponent is always printed.

A user-defined date format is described by the following characters:

Format Symbol	Meaning
"yy"	prints year using two digits.
"yyyy"	prints year using four digits.
"m"	prints month.
"mm"	prints month using two digits.
"mmm"	prints month in an abbreviated string form.
"mmmm"	prints month in its full string form.
"d"	prints day.
"dd"	prints day using two digits.
"ddd"	prints week day in an abbreviated form.
"dddd"	prints week day in its full form.
"/"	prints date separator.
"h"	prints hour.
"hh"	prints hour using two digits.
"n"	prints minutes.
"nn"	prints minutes using two digits.
"s"	prints seconds.
"ss"	prints seconds using two digits.
Prints a time separator.	

Here is a program that demonstrates several examples:

```
PUBLIC SUB Main()
PRINT "-----"
PRINT "User-defined numeric Format examples:"
PRINT Format$(Pi, "-#.###")
PRINT Format$(Pi, "+0#.###0")
PRINT Format$(Pi / 10, "###.# %")
PRINT Format$(-11 ^ 11, "#.##E##")
PRINT "-----"
PRINT "User defined date and time format examples:"
PRINT
PRINT Format$(Now, "mm/dd/yyyy hh:nn:ss")
PRINT Format$(Now, "m/d/yy h:n:s")
PRINT Format$(Now, "ddd dd mmm yyyy")
```

```
PRINT Format$(Now, "dddd dd mmmm yyyy")
END
```

Here is the output:

User-defined numeric Format examples:

3.142
+03.1416
31.4 %
-2.85E+11

User defined date and time format examples:

08/21/2005 21:55:14
8/21/05 21:55:14
Sun 21 Aug 2005
Sunday 21 August 2005

Datatype management

Many times when checking variables, it is important to know what data-type you are dealing with. Gambas provides a host of functions for that purpose. You can check for the following data-types:

IsBoolean / Boolean?	IsByte / Byte?
IsDate / Date?	IsFloat / Float?
IsInteger / Integer?	IsNull / Null?
IsNumber / Number?	IsObject / Object?
IsShort / Short?	IsString / String?

Each of the functions above are called using the standard Gambas language syntax of

```
BooleanResult = IsBoolean? ( Expression )
BooleanResult = IsByte? ( Expression )
BooleanResult = IsDate? ( Expression )
BooleanResult = IsFloat? ( Expression )
BooleanResult = IsInteger? ( Expression )
BooleanResult =IsNull? ( Expression )
BooleanResult = IsNumber? ( Expression )
BooleanResult = IsObject? ( Expression )
BooleanResult = IsShort? ( Expression )
BooleanResult = IsString? ( Expression )
```

and the given function will return TRUE if an expression is of the data-type queried or FALSE if it is not.

TypeOf

TypeOf returns the type of an expression as an integer value. Gambas has defined several predefined constants for the data-types returned by this function. The predefined constants Gambas supports are shown in the table below. The standard Gambas language syntax for this function is:

Type = TypeOf (Expression)

Predefined data-type constants

Data-type	Value
gb.Null	Null value
gb.Boolean	Boolean value
gb.Byte	Byte integer number
gb.Short	Short integer number
gb.Integer	Integer number
gb.Float	Floating point number
gb.Date	Date and time value
gb.String	Character string
gb.Variant	Variant
gb.Object	Object reference

That's about all we will cover in this chapter. In the next chapter, we will return to the Gambas ToolBox and start learning to use some of the more advanced controls, such as iconview, listview, etc. You should now be adequately prepared to deal with more advanced data handling techniques required for those controls.



Chapter 8 – Advanced Controls

In this chapter, we are going to study the a few of the more advanced controls, such as the IconView, ListView, GridView, ColumnView and Tabstrip in order to learn how to use them. For the IconView control, we are going to take a different approach this time, using one of the example programs provided by Gambas. We are going to use the Explorer program written by Gambas creator Benoit Minisini and go through every line of code to understand exactly what is going on in the program and how the IconView control is used.

IconView Control

First of all, let's consider a feature that you have seen before, but you will encounter more frequently in this example. You are familiar with subroutine declarations like:

```
PUBLIC SUB Form_DblClick()  
PUBLIC SUB hButton_Click()  
PUBLIC SUB Combobox_Change()
```

By default, *Name_EventName* is the name of the method called in the event “listener” (a detector built into the Gambas interpreter), when an event is raised. In Chapter 13, you will read more about the “event-driven” programming paradigm, but at this point just note that these subroutine names are not arbitrary. Rather, they conform to the syntax needed to raise built-in Gambas events, where the name of the event is preceded by the underscore character (“_”). In the following code, we will encounter other events, like *_Activate*, *_Rename*, and others, so remember that this syntax is meaningful.

Note: The program “Examples” that are included with Gambas are set to “read-only”. This prevents altering the programs in any way, and prevents us from seeing the Properties setup on the FMain.form page. This can make some of the program operations seem a little mysterious. A fully editable version of the Explorer program has been included in the DVD that accompanies this book – it has been named Explorer1. You may prefer to save that version to disk, and follow it when studying this section. Refer to Appendix B for “jailbreak” instructions to enable editing for any of the programs found in Gambas Examples.

Start Gambas and select the Explorer project from the Miscellaneous section of the Example programs as shown in the following figure. Once you select the Explorer project, the IDE will open up and we will need to get to the code. Do this by double-clicking on the class file or if the form is already up, double-click on a control and then move to the top of the code window. Either way will work as long as you start at the top of the code window where you see the first line:

' Gambas class file

A Beginner's Guide to Gambas – Revised Edition

The first thing we see done in this code is the declaration of the program's global variables. Note that they are prefixed with a '\$' character for clarity in recognizing them. The `$sPath` variable is declared as PRIVATE and used to hold the name of the current file path.



Figure 58: Choosing the Explorer example.

PRIVATE \$sPath AS String

`$bHidden` is a PRIVATE Boolean variable that will act as a flag to be used to determine IF a file is hidden or not. We will use the **Stat** function to check its status. (See Chapter 9 for an explanation of the **Stat** function.)

PRIVATE \$bHidden AS Boolean

`$bCtrl` is a PRIVATE Boolean used as a flag when a CTRL key is pressed. If the user holds down the CTRL key when double-clicking on a folder it will open up in a new window.

PRIVATE \$bCtrl AS Boolean

This is the first subroutine Gambas will execute when the program is run:

STATIC PUBLIC SUB Main()

The next two lines will declare a local variable for the form we want to display to the user and create an instance of the form named *FExplorer*, passing the parameter *System.Home* to a constructor, which is identified as `_new()`, and which takes a string

parameter *sPath* (System.Home) the value of which we obtained from the System class.

```
DIM hForm AS Form  
hForm = NEW FExplorer(System.Home)
```

Now that the form object has been loaded into memory (“instanciated”), show the form:

```
hForm.Show  
END
```

This next subroutine is the constructor that is called when the Main() subroutine is activated and the FExplorer form is instantiated:

```
PUBLIC SUB _new(sPath AS String)
```

“_new” is a special method called when a new object is instantiated. It is used to set the initial conditions of the newly created object. The _new parameter(s) comes directly from the immediately-preceding NEW parameter(s).

We will assign the path passed in (System.Home) as a parameter to our global variable \$sPath:

```
$sPath = sPath
```

Finally, we must call the subroutine RefreshExplorer() to populate the iconview control named ivwExplorer to refresh the display for the new control:

```
RefreshExplorer  
END
```

Here is the RefreshExplorer() subroutine. It is essentially the meat of the program:

```
PRIVATE SUB RefreshExplorer()
```

First, we declare the local variables. Let's start with a string variable to hold file names:

```
DIM sFile AS String
```

Next, declare picture variables for our icon images: parent directory, folder and file:

```
DIM hPictDir AS Picture
```

DIM hPictFile AS Picture

cDir is an array of strings (representing the names of files or directories):

DIM cDir AS NEW String[]

sName is a work string used to represent filenames found in a directory:

DIM sName AS String

This call increments the “Busy” property value that is defined in the Application class (in the Qt library). When this property is set to a value greater than zero, the mouse cursor is locked to a “busy” state to show that the application is busy, and does not respond to user-initiated events. You increment this property at the start of a long job and decrement at the completion of that job, in order to prevent the user from interfering with the job completion.

INC Application.Busy

Once the “*don't interrupt me*” flag is thrown, we set the window title to the system path and call the built-in **Conv\$** function to convert the system charset (*the set of characters the operating system has provided as default*) to what the user has defined as the desktop charset (*the set of characters the user has chosen to see from their desktop*). Note that **Conv** is a synonym for **Conv\$** and you can use either one interchangably.

ME.Title = Conv(\$sPath, System.Charset, Desktop.Charset)

Note that, in the next line of code, “ivwExplorer” is a Control located on our form, which Benoit Minisini created by using the IconView tool from the Qt form toolbox, and so this object belongs to the Gambas class “IconView”. (If you are using Gambas to view the Explorer class file, which is “read-only”, the Properties tab and toolbox are not visible, so it isn't immediately obvious what ivwExplorer is.) Now, in the next line, whatever may exist in the icon view is wiped out by calling the IconView's Clear method:

ivwExplorer.Clear

Next, assign icons to the picture variables we declared. We have an icon to represent folders and one for files:

```
hPictDir = Picture["icon:/48/directory"]
hPictFile = Picture["icon:/48/file"]
```

If the global path variable is not set to the highest level (*i.e., the parent*) indicated by

the "/" string, then we want to create a folder that the user can click to go to the parent directory of the current child. We add this folder to the iconview control named ivwExplorer by using its Add method:

```
IF $sPath <> "/" THEN ivwExplorer.Add("D..", "..", hPictParDir)
```

In the statement above, the .Add method takes three parameters. The first is the name of the key, in this case "D.." which is our key for directories. The second is the text placed below the icon in the iconview, and the final parameter is the name of the variable that is a handle to the filename of the icon we want to display with this entry. Now, we will begin to loop through every file in the current directory to see if it is a hidden file or not and to tag each filename as either a directory or a file.

```
FOR EACH sFile IN Dir($sPath)
```

Gambas initializes strings to null when created, so the very first time we come here, the \$bHidden value will not be TRUE and the IF stmt will be executed in the code below:

```
IF NOT $bHidden THEN
```

This next line of code is a little trickier to decipher. The **Stat** function takes a filename string as its parameter. In this case, the current path held in global variable *\$sPath* is concatenated with the work string *sFile* using the &/ symbol combination. This is a special concatenation symbol used in Gambas specifically to concatenate filenames. The **Stat** function is called with the concatenated filename passed as a parameter and simultaneously the **Stat.Hidden** property is checked. If the **Stat.Hidden** property is set to a TRUE value, the CONTINUE statement executes, forcing program flow to the next iteration of the FOR loop. This whole process basically forces the program to ignore any hidden files it encounters.

```
IF Stat($sPath & sFile).Hidden THEN    'is it hidden?  
    CONTINUE                      'if so, go to next loop iteration  
ENDIF  
ENDIF
```

If we reached this point in the code, the file was not hidden. Now, we will use the built-in file management function **IsDir** to see if the current path and file string (catenated in the Stat call) is a folder or a file. If we have a folder, we will add it to the cDir string array, first tagging it with a letter 'D' for directory or 'F' for file and then appending the filename held in the workstring *sFile* to the 'D' or 'F' tag:

```
IF IsDir($sPath & sFile) THEN    'it was a directory  
    cDir.Add("D" & sFile)  
ELSE  
    cDir.Add("F" & sFile)
```

A Beginner's Guide to Gambas – Revised Edition

```
ENDIF  
NEXT      ' this is the end of the FOR loop...
```

Once everything is loaded in the *cDir* array with the FOR/NEXT loop we call the built-in Sort method and sort the array of strings:

```
cDir.Sort
```

Now that the *cDir* array is sorted, we will loop through the array using the FOR EACH statement which is used specifically for enumerated objects in arrays or collections.

```
FOR EACH sFile IN cDir
```

For every string in the *cDir* array, we will fill the work string *sName* with the name of the file. However, we have to remove the tag first. We will use the **Mid\$** built-in string function to take all the letters of the string after the first (our tag) starting at position 2 in the string:

```
sName = Mid$(sFile, 2)
```

Now, we check the tag of the filename using the string function **Left\$**:

```
IF Left$(sFile) = "D" THEN
```

If the 'D' tag is found, it is a directory and we add a directory to the IconView control using the Add method and our picture of a folder as the last parameter in the call.

```
ivwExplorer.Add(sFile, sName, hPictDir)  
ELSE          ' otherwise it was a file and we add it with a file icon  
    ivwExplorer.Add(sFile, sName, hPictFile)  
ENDIF
```

The following line is commented out in Minisini's code. He set the **.Editable** property to TRUE in the form Properties tab, so this line is redundant here.

```
'ivwExplorer.Item.Editable = TRUE  
NEXT      ' exit the FOR EACH Loop
```

The statements below were commented out but appear to have been used to alter the sort order of the IconView control. We will simply skip over these comments and go to the FINALLY statement:

```
'ivwExplorer.Sorted = FALSE  
'ivwExplorer.Ascending = TRUE
```

A Beginner's Guide to Gambas – Revised Edition

```
'ivwExplorer.Sorted = TRUE
```

FINALLY

```
' this is the last instruction to execute in the subroutine
```

The last thing we need to do is set the Application.busy flag back to normal state:

DEC Application.busy

If an error occurs, we will catch it with the catch clause below:

CATCH

If any error occurs, it should pop up a message box with the error text displayed:

Message.Error(Error.Text)

```
END           ' of our RefreshExplorer routine
```

If you are using Gambas to view the Explorer class file, which is “read-only”, you aren’t able to see the menu bar control on the form. However, you can see the resulting menu bar when you run the program. If the user chooses the quit option from the menu, then this routine executes:

```
PUBLIC SUB mnuQuit_Click()
    ME.Close
END
```

If the user selects the Refresh option from the File menu (visible when you run the program), we will call the RefreshExplorer subroutine described previously.

```
PUBLIC SUB mnuViewRefresh_Click()
    RefreshExplorer
END
```

This subroutine is called when the *activate* event is triggered by a user by double-clicking on a folder in the IconView control:

```
PUBLIC SUB ivwExplorer_Activate()
```

As always, we must declare our local variables. We declare sNewPath as a string to hold the filepath for the target path the user clicked (either a new folder or the root folder). Also, if the control key (CTRL) is pressed when the destination is selected. we declare a new form to show the destination contents, hForm, in a separate window.

```
DIM sNewPath AS String  
DIM hForm AS Form
```

The following IF statement looks at the tag we put on the string to see if the destination is a directory or the root level of the system. If we are at the root level, then we simply return. Otherwise, the code will assign the catenated values of \$sPath and the value held by the **LAST.Current.Key** of the IconView control. This is obtained with the **Mid\$** call starting at position 2 (to bypass our tag character).

```
IF LAST.Current.Key = "D.." THEN  
    IF $sPath = "/" THEN RETURN  
    sNewPath = File.Dir($sPath)  
ELSE  
    sNewPath = $sPath & Mid$(LAST.Current.Key, 2)  
ENDIF
```

We check the newly assigned string value data-type to ensure it is, in fact, a directory and, if it is, we will subsequently check to see if the user held down the control key. Remember, holding down the control key in our program will activate a new window, which is why we declared the local hForm variable.

```
IF IsDir(sNewPath) THEN
```

If the control key was held down, we will toggle the value back to FALSE before instantiating a new window. Then, we will move our control to be offset 16 pixels right and below the current window, having the same height and width using the hForm.Move method. Finally, we show the form and refresh the Explorer with our RefreshExplorer subroutine.

```
IF $bCtrl THEN  
    $bCtrl = FALSE  
    hForm = NEW FExplorer(sNewPath)  
    hForm.Move(ME.X + 16, ME.Y + 16, ME.W, ME.H)  
    hForm.Show  
ELSE
```

Othewise, the control key was not held down so we simply assign the sNewPath value to the global \$sPath and refresh the explorer:

```
$sPath = sNewPath  
RefreshExplorer  
ENDIF  
ENDIF  
END
```

A Beginner's Guide to Gambas – Revised Edition

When the user clicks this routine, it calls our subroutine to show the hidden files or, if they are showing, turn off viewing of hidden files:

```
PUBLIC SUB mnuViewHidden_Click()
    ToggleViewHidden
END
```

If the user chooses to show hidden files in the explorer by selecting the Show Hidden Files option, then this next routine is executed. The first line checks the value of the mnuViewHidden.Checked property and toggles the value by essentially making \$bHidden become whatever the logical NOT value of mnuViewHidden.Checked is at the moment it is checked. It then assigns the opposite value to the property and refreshes the view by calling RefreshExplorer. The result is that the hidden files are shown.

```
PRIVATE SUB ToggleViewHidden()
    $bHidden = NOT mnuViewHidden.Checked
    mnuViewHidden.Checked = $bHidden
    RefreshExplorer
END
```

This subroutine is executed when the menu's about item is clicked. It simply displays a message box crediting the code to the Gambas founder, B. Minisini:

```
PUBLIC SUB mnuAbout_Click()
    Message("IconView example written by\nBenoît Minisini")
END
```

The next subroutine is executed when the user single-clicks on an item in the explorer and opts to rename the item. As it was originally written, it does not check to see if the rename actually changed the item. For example, if the user pressed ESC, the item would revert to its original name.

```
PUBLIC SUB ivwExplorer_Rename()
    Message("'" & Mid$(LAST.Item.Key, 2) & "' has been renamed to '" & LAST.Item.Text & "'")
END
```

We can remedy this by changing the code to read like this:

```
PUBLIC SUB ivwExplorer_Rename()
IF Mid$(LAST.Item.Key,2) <> Last.Item.Text THEN
    Message("'" & Mid$(LAST.Item.Key,2) & "' has been renamed to '" & LAST.Item.Text & "'")
ELSE
    Message("'" & Mid$(LAST.Item.Key,2) & "' was left unchanged.'")
ENDIF
```

END

The final two subroutines toggle the global variable \$bCtrl whenever the control key is pressed or released.

```
PUBLIC SUB ivwExplorer_KeyPress()
  IF Key.Control THEN $bCtrl = TRUE
END
PUBLIC SUB ivwExplorer_KeyRelease()
  IF Key.Control THEN $bCtrl = FALSE
END
```

There you have it. Everything you need to know about using the IconView control in a cool application. Next, we will look at the ListView control.

ListView Control

The **ListView** control inherits Control and implements a list of selectable text items with icons. ListView items are indexed by a key. They display a string and an icon. This control has an internal cursor used for accessing its items. You must use the **Move** methods (MoveAbove, MoveBelow, MoveCurrent, MoveFirst, MoveLast, MoveNext, MovePrevious, MoveTo) to move the internal cursor, and you need to use the Item property to get the text of the item the cursor points at. This class is creatable and the standard calling convention is:

```
DIM hListView AS ListView
hListView = NEW ListView ( Parent AS Container )
```

The above code creates a new ListView control that acts like a read-only array. Let's create a program that will implement the ListView control. Create a new project named ListView that is a Qt graphical user interface type project. When the IDE appears after you go through the project wizard, create a simple form that looks like the figure below.

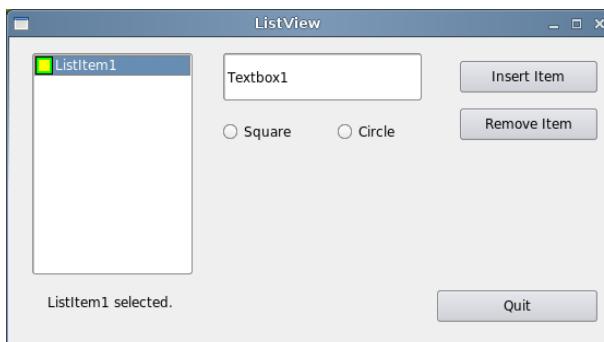


Figure 59: Layout for our ListView example.

A Beginner's Guide to Gambas – Revised Edition

From the left top of the picture, you have our ListView control named ListView1, a Textbox named Textbox1, and the Insert Item button named Button1. There are two radiobuttons, named RadioButton1 and RadioButton2 and the Remove Item button named Button2. At the bottom of the form is a TLabel named TLabel1 and the Quit button named Button3.

Once you have created the form and made it a startup class, we need to add the following code to the code window to see how to implement a ListView control:

```
' Gambas class file
sStatus AS String

PUBLIC SUB Form_Open()
    DIM picSquare AS NEW Picture
    picSquare = picSquare.Load ("square.png")
```

In the above, we used the Picture class variable "picSquare" in order to return a value (a bitmap) using the "Load" method. The variable "picSquare" then accepts the returned value (the bitmap). Syntax: Picture.Load(filename) Currently, Picture can accept .png, .jpg, .bmp, .gif, and .xpm file formats. Actually, if you dimension a Picture variable with any other name, that also works. For example: DIM picXYZ AS NEW Picture, which could then be used in the statement: picSquare = picXYZ.Load("square.png"). It is merely for convenience that we use picSquare.Load in the code, since we have already dimensioned picSquare as a Picture class variable.

The following lines will add an item to the ListView with a starting entry

```
ListView1.Add ("List1Item","List1Item", picSquare)
TextLabel1.Text = ListView1.Item.Text
ListView1_Click
END
```

When the program begins and the form is first displayed on-screen, the routine above is executed. We create two local variables for our icons that will be used in the list and load them into memory. Next, we add a default key and item to the list using the ListView1.Add method, specifying the picSquare icon to be associated with this item. Don't worry about the icons for now – we will create them last. Next, we need to create an event to refresh the list anytime an item is added or it is clicked on by the user.

```
PUBLIC SUB ListView1_Click()
    ListView1.Item.Selected = TRUE      'First, identify the item as selected.
    ListView1.MoveCurrent             'Then, move the cursor to the selected item.
    TextLabel1.Text = ListView1.Item.Text & sStatus
END
```

A Beginner's Guide to Gambas – Revised Edition

In the ListView1_Click subroutine, the first line of code identifies the current item as “selected” by setting the Item.Selected to TRUE, then moves the internal cursor to the most current item. This way, when you add any new items, they will automatically be highlighted. Finally, the third line updates our TextLabel1.Text value with the text of the current (or newly added) selection and the current status (held in global var sStatus AS String) appended. If the user clicks on the Insert item button (*we named it Button1*), this next routine is executed:

PUBLIC SUB Button1_Click()

Declare a local variable, picToUse, for our icon to be displayed:

DIM picToUse AS NEW Picture

Next, check which radioButton has been clicked. If the first button has been clicked, we will load the image for square.png, otherwise, circle.png is loaded.

```
IF Textbox1.Text <> NULL THEN  
    IF RadioButton1.Value THEN  
        picToUse = picToUse.Load("square.png")  
    ELSE  
        picToUse = picToUse.Load("circle.png")  
    END IF
```

Now we will add a new entry with a key and name in the text box:

```
Listview1.Add(Textbox1.Text,Textbox1.Text,picToUse)
```

This empties out textbox:

```
TextBox1.Text = ""  
sStatus = " current." 'set status to "current"
```

This call to the ListView1_Click subroutine will update (refresh) our ListView control:

```
Listview1_Click
```

Next, we must ensure the new item is in the visible area of the control:

```
Listview1.Item.EnsureVisible  
END IF  
END
```

The Button2_Click subroutine is called whenever the user decides to delete the currently selected item in the ListView control.

PUBLIC SUB Button2_Click()

This next line of code gets out cursor lined up to the current selection. The MoveCurrent method moves the internal cursor to the current item. It returns a TRUE value if there is no current item, in which case we will simply return as there is nothing to delete:

IF ListView1.MoveCurrent() THEN RETURN

If we reached this point, we have moved the cursor to and need to remove the current cursor item:

ListView1.Remove(ListView1.Item.Text)

Clean up our TextLabel1.Text display with this call:

TextLabel1.Text = ""

Now, we need to update the cursor position to the new current item (since we are now pointed at a deleted item). But first, we need to check the .Count property to see if we have just deleted the last item in the list. If the count is greater than zero, then we have items left in the list. Otherwise, the IF statement will not be true and the code will be bypassed:

**IF ListView1.Count > 0 THEN
 ListView1.MoveCurrent**

After the .Remove method is executed, the current item becomes the item below the removed item in the list, unless the removed item was the lowest-down item on the list -- in that case, the current item becomes the new lowest down item on the list. This determines the new position of the internal cursor established by .MoveCurrent.

ListView1.Item.Selected = TRUE 'Selects the current item

Note that for removal of items, the order of .MoveCurrent and .Item.Selected must be reversed, compared to the ListView1_Click subroutine.

**sStatus = " selected."
 ListView1_Click** 'this will force an update of the TextLabel text.
**END IF
END**

If the user clicks their mouse on an item in the ListView control, this routine is called. We will simply update the TextLabel1.Text property with the text of the item selected and refresh the ListView control by calling our ListView1_Click subroutine.

```
PUBLIC SUB ListView1_Select()
    TextLabel1.Text = ListView1.Item.Text
    sStatus = " selected."
    ListView1_Click
END
```

If the user double-clicks their mouse on an item in the ListView control, it will raise the `_Activate` event, indicating the user has picked this item for some action to occur. In this case, we will simply update the `TextLabel1.Text` property with the text of the item selected, appended with the word “*activated*”, and refresh the ListView control by calling our `ListView1_Click` subroutine.

```
PUBLIC SUB ListView1_Activate()
    TextLabel1.Text = ListView1.Item.Text & " activated."
    sStatus = " activated."
    ListView1_Click
END
```

Note that the `_Activate` event is triggered by a double-click, just like `DblClick`. However, `DblClick` is triggered ANYWHERE on your control, including your items, scrollbars, buttons, or any part of your control. Therefore, if you want to enable users to double-click an item in a list, as in ListView, use `_Activate`.

If the user clicks their mouse on the `Button3` (Quit) button, this routine is called. We want to exit cleanly so we invoke the `close` method for the form using the `ME.Close` call.

```
PUBLIC SUB Button3_Click()
    ME.Close
END
```

Using the GB Icon Edit Tool

As you may already be aware, an icon is a small pictogram used in graphical user interfaces to supplement the presentation of textual information to the user¹³. Modern computers can handle bitmapped graphics with ease, so icons are widely used to assist users. Icons were first developed as a tool for making computer interfaces easier for novices to grasp in the 1970s, at the Xerox Palo Alto Research Center facility. Icon-driven interfaces were later popularized by the Apple Macintosh, the Amiga and the Microsoft Windows operating environments.

A computer icon usually ranges from 16 by 16 pixels up to 128 by 128 pixels. Some operating systems feature icons up to 512 by 512 pixels. Icon size can be related to the size of

13 Refer to http://en.wikipedia.org/wiki/Computer_icon for more information about icons.

the graphical output device or to the amount of information that must be displayed. Vision-impaired users (due to such conditions as poor lighting, tired eyes, medical impairments, bright backgrounds, or color blindness) may need the ability to use larger icons.

At this point, all the code is created for our ListView example program. We still need to create the circle.png and square.png icons. We will use the GB icon editor to do that. To bring up the icon editor, we will go to the Project Window in the IDE. In the Project TreeView, find the Data folder and right-click your mouse. Pick the New item and the Image subitem. You will see this dialog pop up:

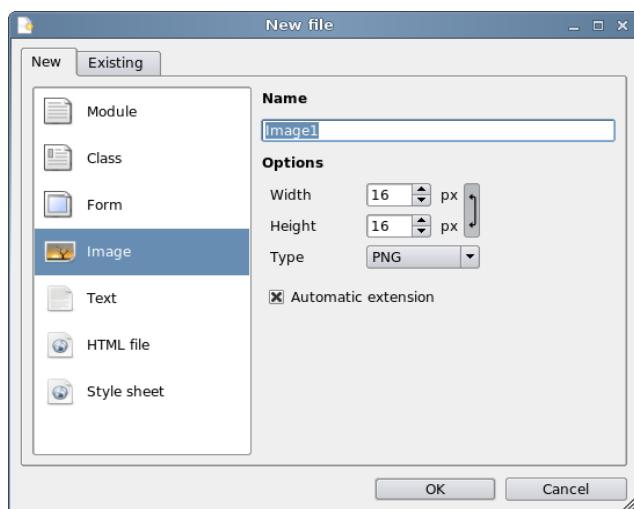


Figure 60: Creating a new Icon image in GB.

Simply type “square” in the Name field and click ok. Now, the icon editor appears and you can create an icon. Use the rectangle option from the IconEditor Toolbox and create a blue square and save it with the disk-shaped icon.



Figure 61: The Icon Editor Toolbox.

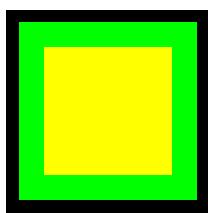


Figure 62: Our square icon image.

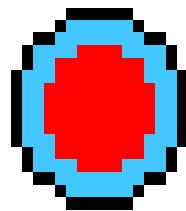


Figure 63: Our circle icon image.

Repeat this process, creating another icon named “circle.png” and save that too. That is it! Your program is ready to run. Try it out and see how the ListView control works.

The TreeView Control

The TreeView control works almost identically to the ListView control. The chief difference is that the TreeView supports nested “children” and allows you to traverse from an inner (child) level outward to the parent of the child, all the way out to the top of the tree. This control adds a few methods specifically for the purpose of managing the traversal of the parent/child trees. TreeView, like all other controls, inherits its attributes from the Control class. This control implements a tree view of selectable text items with icons.

Tree view items are indexed by a key. They display a string and an icon for each item. This control has an internal cursor used for accessing its items. Use the Move methods (Move, MoveAbove, MoveBack, MoveBelow, MoveChild, MoveCurrent, MoveFirst, MoveLast, MoveNext, MoveParent, MovePrevious, MoveTo) to move the internal cursor, and the Item property to get the item it points at. This class is creatable. The standard calling convention is:

```
DIM hTreeView AS TreeView  
hTreeView = NEW TreeView ( Parent AS Container )
```

The code above creates a new TreeView control. This class acts like a read-only array:

```
DIM hTreeView AS TreeView  
DIM hTreeViewItem AS .TreeViewItem  
hTreeViewItem = hTreeView [ Key AS String ]
```

The line of code above will returns a TreeView item from its key. The internal cursor is moved to the current item. Now, let's use an example program from the DVD included with this book. Start Gambas and select the TreeView1 sample program (this is based on the TreeView example from Gambas). When the IDE opens up, you should see this:

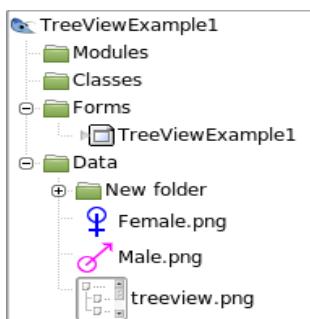


Figure 64: Treeview example.

A Beginner's Guide to Gambas – Revised Edition

Next, double-click on the TreeView1 form and bring up the form. Double click on the form to open up the code edit window and you will see the following listing, which we will dissect line by line to understand how it works:

```
' Gambas class file
PUBLIC intEventNumber AS Integer
```

The global variable intEventNumber is used to track the number of events that occur for our event stack. Each time we process an event, we will increment this variable. We declare two local Picture variables, picMale and picFemale and load them into memory for the program by using the Picture.Load method.

```
PUBLIC SUB Form_Open()
DIM picMale AS NEW Picture
DIM picFemale AS NEW Picture
picFemale.Load("Female.png")
picMale.Load ("Male.png")
```

'This will populate our treeview with our starting entries
'Note: just keep the entries text and their keys the same to keep it simple

Next, we will add the starting values of our TreeView control. We will add four items to the control. Each item key will be the same as the item name. The items Ted and Sally will be children of item Bill, while items Frank and Barbara will be children of Sally.

```
TreeView1.Add ("Bill","Bill", picMale)           'Bill has no parent (in our TreeView), so no 4th parameter.
TreeView1.Add ("Ted","Ted",picMale,"Bill")
TreeView1.Add ("Sally","Sally",picFemale,"Bill")
TreeView1.Add ("Frank","Frank",picMale,"Sally")
TreeView1.Add ("Barbara","Barbara",picFemale,"Sally")  
The syntax is: TreeView.Add("Key", "Text", Picture[name], "Parent Key")
```

In order to see the full TreeView after opening the program, we indicate that the nodes should be expanded by default:

```
TreeView1["Bill"].Expanded = TRUE
TreeView1["Sally"].Expanded = TRUE
```

```
END
```

(NOTE: the Gambas Wiki documentation for this property lists it as *Expand*, not *Expanded*. Using *Expand* will NOT work.)

The next subroutine updates the TextLabel so that we know how many children an

entry has. We find out whether the item clicked is a parent (determined by whether or not it has children). If the number of children is greater than one, we want our label (TextLabel1.Text) to use the plural form of child. If there is only one child, our label will say child, and if there are none, we want it to say the item has no children. We use the IF THEN/ELSE statement to make this check:

```
PRIVATE SUB RefreshInfo()
DIM sText AS String
IF NOT TreeView1.Current THEN      'Declared PRIVATE to protect sText from being changed elsewhere
    TextLabel1.Text = ""
RETURN
ENDIF
```

'if a current item does not exist,
 'clear the TextLabel box.
 'and exit the subroutine.

In the following text block, note that the properties .Children, .Text, .X, .Y, .W, and .H can all be read without placing "TreeView1" in front. Since, by default, they read the properties of the current item, it is not necessary to type TreeView1.Text, etc. However, you CAN optionally type the long form, if you want to make this explicit.

```
WITH TreeView1.Current THEN
    IF .Children > 1 THEN
        sText = .Text & " has no children."
    ELSE IF .Children = 0 THEN
        sText = .Text & " has 1 child."
    END IF
    'In the next line, "sText &= ..." is equivalent to "sText = sText & ...."
    sText &= "<br>Item rect is (" & .X & "," & .Y & "," & .W & "," & .H & ")"
    '<br> is HTML for "line break".
```

As previously discussed, a TextLabel control can accept HTML formatting.

```
TextLabel1.Text = sText
END
```

If the user enters data in the Textbox control and clicks the Insert Name button (*Button1*), the following click-event is executed. It first declares two local variable. The sIcon variable will determine which picture to load based on which RadioButton was clicked at the time the Insert Name button was clicked. The string variable, sParent, is assigned the value of the current item's key. If the MoveCurrent method tries to move the internal cursor to the current item and there is no item, it returns TRUE. Otherwise, FALSE will be returned and we assign the sParent string the value of the key that is current.

```
PUBLIC SUB Button1_Click()
DIM sParent AS String
IF Textbox1.Text <> NULL THEN
```

```
IF RadioButton1.Value THEN
    sIcon = "Male.png"
ELSE
    sIcon = "Female.png"
ENDIF
'Gets the parent item: the current item, or nothing if the treeview is void
sParent = TreeView1.Key
```

Now, we need to add the new entry with a key and a name of what was in the text box. We will place it in the TreeView control as a child of the currently selected entry. Note that the key names must be unique or it will crash the program. We could use the Exist method to find out if a key exists with the name Textbox1.Text before making the call to the Add method, but that was not done here. If we did that, the code would make a check similar to this:

```
IF Exist(Textbox1.Text) <> TRUE THEN
    TreeView1.Add(Textbox1.Text, Textbox1.Text, sIcon, sParent)
ENDIF
```

However, the code in the program simply forces the Add method to take the Textbox1.Text and put it in the list:

```
TreeView1.Add(Textbox1.Text, Textbox1.Text, Picture[sIcon], sParent).EnsureVisible
TextBox1.Text = ""      'This empties out textbox
RefreshInfo          'This updates the label, and reflects the new number of kids.
ENDIF
END
```

The call to EnsureVisible will make sure that the item we just added to the list is in the visible area of the control. If necessary, the control will scroll so the item is visible. Rather than sharing a line with the .Add method, the same command could also be used on a separate line:

```
TreeView1.Item.EnsureVisible
```

If the user wants to remove a name from the TreeView by clicking the Remove Name button (*Button2*) then this subroutine is executed:

```
PUBLIC SUB Button2_Click()
```

First of all, we must get the cursor lined up to our current selection and make sure that the current item isn't an empty or null value. If the MoveCurrent method returns a true value, the list was empty and we are done. The code invokes the RETURN call and we jump back to the calling process. Otherwise, whatever the current item is will be removed by calling the TreeView1.Remove method in the second line, below:

```
IF TreeView1.MoveCurrent() THEN RETURN
```

```
'Lets remove the current cursor item
```

It is necessary to de-select the current item before deleting it, or else your deletions will leave “ghost items” in the TreeView, and also disorder the icons. The current item is still current, even when it is de-selected, and you don't have to name another item as current – the Gambas interpreter will handle that. Therefore, use the next line to de-select the item being removed, then use the line after that to remove it.

```
TreeView1.Current.Selected = FALSE
```

```
TreeView1.Remove(TreeView1.Item.Text)
```

Now we must move the cursor to the current item (since we are now pointing at a deleted item). Before we do that we need to check the count property to make sure we didn't delete the last item in the list. If we did, then we obviously don't run this part of the code. The IF statement checks for a count greater than zero and, if TRUE, will move to the current item, select it so the cursor is highlighted, and update the control with a call to the TreeView1_Click subroutine.

```
IF TreeView1.Count > 0 THEN
```

```
    TreeView1.MoveCurrent
```

```
'This selects or 'highlights' our current item
```

```
    TreeView1.Item.Selected = TRUE
```

```
'This will update our label and reflect the new number of kids
```

```
    RefreshInfo
```

```
    TreeView1_Delete      'This will update the event stack.
```

```
END IF
```

```
END
```

If the user clicks on the minus icon in the TreeView control, it sets off a Collapse event. The routine below is called. It simply updates the event stack, as described previously, and increments the event counter, intEventNumber, by one.

```
PUBLIC SUB TreeView1_Collapse()
```

```
'This just updates our event stack
```

```
Stack_Event_Log("Collapse")
```

```
END
```

If the user double-clicks on an item in the TreeView control, it sets off a Dbl_Click event. It is basically the same as an Activate event. The routine below is called. It simply updates the event stack, as described previously, and increments the event counter, intEventNumber, by one.

A Beginner's Guide to Gambas – Revised Edition

```
PUBLIC SUB TreeView1_DblClick()
```

```
'This just updates our event stack
```

```
Stack_Event_Log("DblClick")
```

```
END
```

```
PUBLIC SUB TreeView1_Select()
```

```
'This just updates our event stack
```

```
RefreshInfo
```

```
Stack_Event_Log("Select")
```

```
END
```

```
PUBLIC SUB TreeView1_Delete()
```

```
'This just updates our event stack
```

```
Stack_Event_Log("Delete")
```

```
END
```

If the user clicks on the plus icon in the TreeView control, it sets off an Expand event. The routine below is called. It simply updates the event stack.

```
PUBLIC SUB TreeView1_Expand()
```

```
'This just updates our event stack
```

```
Stack_Event_Log("Expand")
```

```
END
```

```
PUBLIC SUB Button3_Click()
```

```
TextArea1.Text = ""
```

```
END
```

The **Help | About** menu item generates a click event that calls this next subroutine. It simply gives credit to the original authors, C. Packard and Fabien Hutrel.

```
PUBLIC SUB About_Click()
```

```
Message.Info ("TreeView example written by C. Packard and Fabien Hutrel.\nModified and renamed by Jon Nicholson")
```

```
END
```

Note that in the above text, you can use the “\n” control code to make a line break in a MessageBox. Alternatively, you could break the string apart and concatenate with &Chr(10), which also executes a line break.

If the user double-clicks on an item in the TreeView control, it sets off an Activate event. It is similar to a DblClick event in that _Activate is triggered by double-clicking, just like _DblClick is. However, the _DblClick event is raised everywhere in the control, even on internal scrollbars, buttons, etc. Use _Activate when you want to limit the clickable area to specific item choices in a control, as in ListView and TreeView. When the routine below is

called, it simply updates the event stack.

```
PUBLIC SUB TreeView1_Activate()
'This just updates our event stack
Stack_Event_Log("Activate")
END
```

This next routine is dead code. It never gets executed because nothing creates an event to get here. To use it effectively, a menu item, Rename, should be added to the Help menu and it should be called from the MenuItem.Click event. If this is done, then the event would be generated and you could write a bit of code to get the new name from the user and change it. The event stack would be properly updated, first with the rename event and then with the click event.

```
PUBLIC SUB TreeView1_Rename()
'This just updates our event stack
Stack_Event_Log("Rename")
END
```

Every subroutine uses the following sub to update the event stack. The string variable “anevent” is declared in the sub declaration and receives the string parameter passed by the calling subroutines, above.

```
PRIVATE SUB Stack_Event_Log(anevent AS String)
'This updates our event stack and displays the current node key.
DIM sKey as String
```

In the next line, the meaning of the syntax "TRY Statement" is that TRY tries to execute a statement, without raising an error if the attempt fails.

```
TRY sKey = TreeView1.Item.Key
```

In the next line, note that the new event is concatenated with the existing TestArea1 box contents, but that a line break (\n) is included so that each new event will be on its own line at the top of the stack.

```
TextArea1.Text = "Event(" & intEventNumber & "): " & anevent & " Item: ' & sKey & "\n" & TextArea1.Text
intEventNumber = intEventNumber + 1
END
```

That is all there is to the TreeView program. You can play around with the code and implement the suggestions made herein. Once you are able to understand how to use these controls effectively, your GB interfaces will become more sophisticated. Next, we will take a look at the last two advanced view controls, the GridView and ColumnView controls.

The GridView Control

The GridView control, like all the other controls in the toolbox, inherits its attributes from the Control class. GridView implements a control that displays data in a grid. This class is creatable. Standard calling convention for GridView is:

```
DIM hGridView AS GridView
hGridView = NEW GridView ( Parent AS Container )
```

The code above will create a new GridView. This class acts like a read-only array. To retrieve the contents of a cell in the grid, use this code to declare a GridCell variable that can read the GridView array (the GridCell is a virtual class):

```
DIM hGridView AS GridView
DIM hGridCell AS .GridCell
hGridCell = hGridView [ Row AS Integer, Column AS Integer ]
```

The line above returns a cell from its row and its column. Our example program for this control will create a grid that is four rows of three columns. We will populate the first three rows with text and the last row with pictures. We will take the last row of text and demonstrate how the alignment property works for text data. Our program will have three buttons, Quit, Clear, and Reset.

The Quit button will simply close the program. Clear will invoke the GridView.Clear method to erase the contents of the grid and display empty grid cells. The Reset button will repopulate the contents of the grid with the data the program started with (i.e., the data populated when the form's constructor method was called. Our program will look like this:

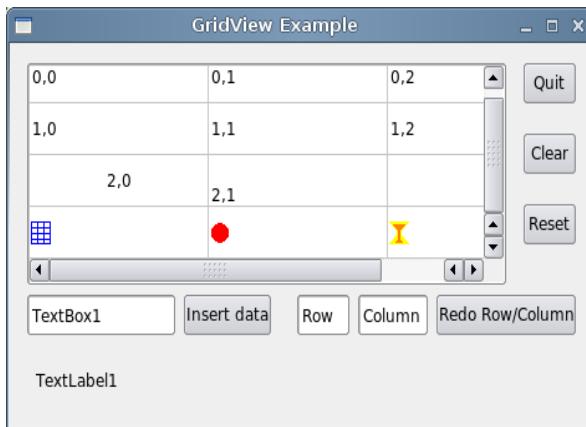


Figure 65: What our GridView will look like.

To begin, start Gambas and select the New Project option, choosing to create a QT project. Make the project translatable and form controls public by checking the two boxes in the Wizard. Name the project GridView, and give it the “title” of “GridView Example”.

When the IDE appears, we will need to go to the project window and bring up the ICON Editor. We will create three icons, named GridPic, GridPic1, and GridPic2, as shown above. Use the 16 pixels by 16 pixels setting, and try to get them reasonably close to what you see in the picture above if you can.

Next, create a new startup class form and put the GridView control, named GridView1 on the form as shown above. Then add the three buttons to the right of the GridView, named “Quit” (Button1), “Clear” (Button2), and “Reset” (Button3) to the form. Create three TextBoxes underneath the GridView, and use the Properties tab to set the default text on TextBox2 to “Row” and on TextBox3 to “Column”. Add two more buttons beneath the GridView, and name them “Insert data” (Button4) and “Redo Row/Column” (Button5). Finally, add a TextLabel control at the bottom of the form, named TextLabel1. Now, we have the form designed and it looks like what we expect.

Double-click on the form, but not on a control, and the Form_Open subroutine will display in the code window. We will set the caption to your program here:

```
' Gambas class file
PUBLIC SUB Form_Open()
    FMain.Caption = " GridView Example "
END
```

When the program starts, if a constructor exists, it is executed first. Our constructor will load our three icons and populate the grid. First, declare three local variables for the pictures, hPic1, hPic2, and hPic3, as shown below:

```
PUBLIC SUB _new()
    DIM hPic1 AS Picture
    DIM hPic2 AS Picture
    DIM hPic3 AS Picture
```

Now, we must instantiate the picture variables and load the pictures.

```
hPic1 = NEW Picture
hPic1 = hPic1.Load("GridPic1.png")

hPic2 = NEW Picture
hPic2 = hPic2.Load("GridPic2.png")

hPic3 = NEW Picture
hPic3 = hPic3.Load("GridPic3.png")
```

The next thing we need to do is define the dimensions of the grid. Our grid will have three columns and four rows, so we set the Columns and Rows properties using their virtual class gridColumn and gridrow property, count to set the rows and columns as shown below:

```
GridView1.Columns.Count = 3  
GridView1.Rows.Count = 4
```

We can also define the width and height dimensions of the grid columns and rows. Bear in mind that it is possible to set different height values for every row and different width values for every column. We will set the the W and H properties for our rows and columns as shown below:

```
GridView1.Columns.Width = 144  
GridView1.Rows.Height = 36
```

Next, we will populate the grid with some data, simply identifying the row,column position with text:

```
GridView1[0,0].Text = "0,0"  
GridView1[0,1].Text = "0,1"  
GridView1[0,2].Text = "0,2"
```

```
GridView1[1,0].Text = "1,0"  
GridView1[1,1].Text = "1,1"  
GridView1[1,2].Text = "1,2"
```

For the last text row, we want to demonstrate how to use the Alignment property:

```
GridView1[2,0].Alignment = Align.Center  
GridView1[2,1].Alignment = Align.BottomLeft  
GridView1[2,2].Alignment = Align.TopRight
```

After setting the alignment properties for each cell in the row, we will add text:

```
GridView1[2,0].Text = "2,0"  
GridView1[2,1].Text = "2,1"  
GridView1[2,2].Text = "2,2"
```

Finally, we will add our three icons to the grid. Note that, if we wanted to, we could also add text to the same cells – the two datatypes can exist and be displayed in a cell at the same time.

```
GridView1[3,0].Picture = hPic1  
GridView1[3,1].Picture = hPic2  
GridView1[3,2].Picture = hPic3  
END
```

The constructor is complete. Now, double-click on the Quit Button to get the Button1_Click event subroutine, where we will add our Me.Close call to shut down the

program:

```
PUBLIC SUB Button1_Click()
ME.Close
END
```

The Clear button is next. Double-click on the Clear Button to get the Button2_Click event subroutine, where we will add our code to reset the grid to blank values:

```
PUBLIC SUB Button2_Click()
GridView1.Clear
END
```

The easiest way to repopulate the data is to simply invoke the constructor once again. Add this code so that is what we will do.

```
PUBLIC SUB Button3_Click()
_new
END
```

Now, allow the user to edit text contained in the cells. Double-click on the “Insert data” button (Button4) and enter this code:

```
PUBLIC SUB Button4_Click()
DIM EnteredData AS String
DIM Row AS Integer
DIM Column AS Integer
```

Convert whatever row/column numbers the user types into the TextBoxes into integers.

```
Row = CInt(TextBox2.Text)
Column = CInt(TextBox3.Text)
```

If the user exceeds the permitted values for either the row index or the column index, terminate the subroutine.

```
IF Row > 3 OR Column > 2 THEN
    TextLabel1.Text = "<b>Sorry, Rows must be 0-3 and Columns must be 0-2</b>"
    RETURN
ENDIF
EnteredData = TextBox1.Text
GridView1[Row, Column].Text = TextBox1.Text
END
```

Allow the user to click one button to empty the TextBoxes of the disallowed coordinates and also empty the error message in the TextLabel. Double-click on the “Erase Row/Column” button (Button 5) and enter this code:

```
PUBLIC SUB Button5_Click()
    TextBox2.Text = ""
    TextBox3.Text = ""
    TextLabel1.Text = ""
END
```

Finally, when the user clicks around in the various cells, the text contents and cell coordinates will reflect in the TextBoxes. Double-click on the GridView control on the form, and enter this code:

```
PUBLIC SUB GridView1_Click()
    DIM Row AS Integer
    DIM Column AS Integer
    TextBox1.Text = GridView1.Current.Text
    TextBox2.Text = GridView1.Row
    TextBox3.Text = GridView1.Column
END
```

Our program is complete. Run it and see how it works. After you are satisfied that it works as advertised, close it down and next let's learn about the ColumnView control next.

The ColumnView Control

For our next example, we are going to create a program that displays data in a ColumnView, as shown below:

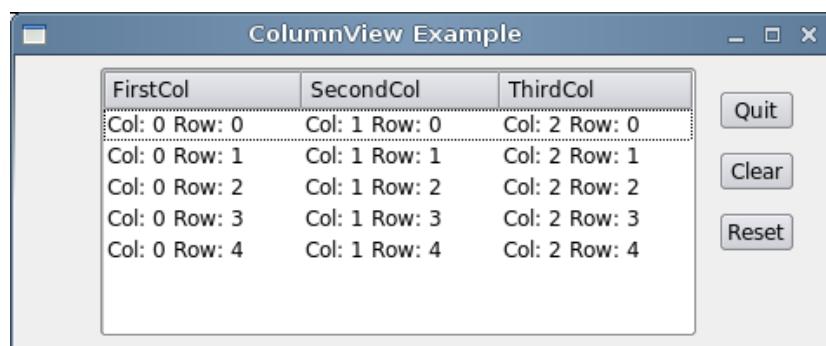


Figure 66: Our ColumnView example.

Start a new project as a Qt graphical interface program, and name it ColumnView. When the IDE starts, open the FMain form, and use the toolbox to set up the ColumnView control and the three button controls according to the figure above. Our simple example will create three columns and

A Beginner's Guide to Gambas – Revised Edition

five rows. We will populate the column data with the row,column id of the particular item. To do this, we will need to set the width of each column to be dynamically calculated based on the width of the control and the number of columns specified. We will do this in the constructor. When the program starts up, all we are going to do on the Form_Open call is set the window caption, as shown in the Form_Open() subroutine.

```
' Gambas class file
PUBLIC SUB Form_Open()
    Form1.Caption = " ColumnView Example "
END
```

The constructor method is where all the work to populate the control is done. We need to declare three local integer variables, iwidth to hold the value of the calculated column width, irowcounter and icolcounter, which are used for our loop structures.

```
PUBLIC SUB _new()
    ' declare our local vars
    DIM iwidth AS Integer
    DIM irowcounter AS Integer
    DIM icolcounter AS Integer

    'set the number of columns to 3
    ColumnView1.Columns.Count = 3
    'calculate the column width based on the number of columns set
    iwidth = ColumnView1.Width / ColumnView1.Columns.Count
    'for the first column, set the column width and add a column title
    ColumnView1.Columns[0].Width = iwidth
    ColumnView1.Columns[0].Text="FirstCol"
    'for the second column, set the column width and add a column title
    ColumnView1.Columns[1].Width = iwidth
    ColumnView1.Columns[1].Text="SecondCol"
    'for the third column, set the column width and add a column title
    ColumnView1.Columns[2].Width = iwidth
    ColumnView1.Columns[2].Text="ThirdCol"
```

Now, we need to set up the outer (row) and inner (column) loops. Basically, we will loop through every every row in the control and, at each row, we will loop through each column, populating the ColumnView1[row][column] array items with a dynamically built string of text that will represent the current position within the inner and outer loop.

```
FOR irowcounter = 0 TO 4 'for all five of our rows
    'call the Add method with [row][col] to add to ColumnView1 control
    ColumnView1.Add(irowcounter, icolcounter)' start outer (row) loop
```

```
FOR icolcounter = 0 TO ColumnView1.Columns.Count - 1
    'add text to the [row][col] item
    ColumnView1(irowcounter)(icolcounter) = "Col: " & icolcounter & " Row: " & irowcounter
NEXT 'column
NEXT 'row
END 'our constructor
```

The Quit Button here works just like in the previous example. To get the Button1_Click event subroutine, double-click and add the Me.Close call to shut down the program:

```
PUBLIC SUB Button1_Click()
    ME.Close
END
```

The Clear button is next. Double-click on the Clear Button to get the Button2_Click event subroutine, where we will add our code to reset the grid to blank values:

```
PUBLIC SUB Button2_Click()
    ColumnView1.Clear
END
```

For our Reset button, the easiest way to repopulate the data is to simply invoke the constructor once again. Add this code to accomplish that.

```
PUBLIC SUB Button3_Click()
    _new
END
```

That is all there is to building and populating the ColumnView control. There are several events that can be used to enhance the functionality of our program. For example, if a user clicked on column 2 of row three, we could use the _Activate event to allow the user to change the data for that item. One method of accomplishing that is to modify the form as shown in the figure below.

Next, add a TLabel to the bottom-left of the ColumnView.control. Place a Label underneath it, with the text set as “Row”. Place three TextBoxes underneath the ColumnView, with three Labels underneath them, with the text set as Columns 1, 2 and 3. Place a Button to the lower right of the ColumnView, and set the text as “Update”.

A Beginner's Guide to Gambas – Revised Edition

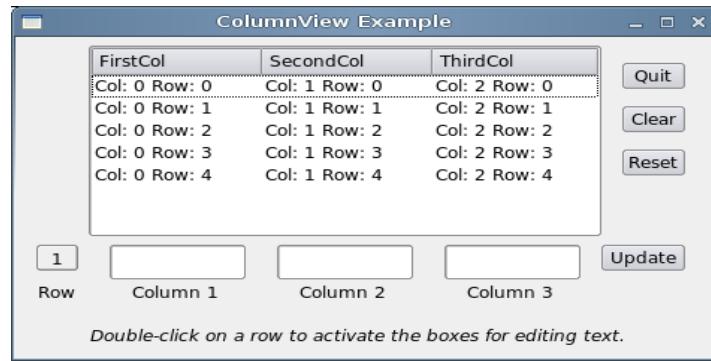


Figure 67: Our editable ColumnView example.

Go to the top of the FMain.class page and insert the following code immediately after the **PUBLIC SUB FORM_OPEN()...END BLOCK:**

sKey as STRING 'A global variable for writing to the TextLabel.
Flag AS Boolean 'This prevents incomplete data entry.

Next, go to the line that reads “**END 'our constructor'**”. In an empty line immediately above that, type

InitialTextBox

Go to the bottom of the code you have already input, and append this code to it:

```
' Start of additional code
PUBLIC SUB InitialTextBox()
    sKey = ColumnView1.Current.Key
    TextLabel1.Text = CInt(sKey) + 1
    TextBox1.Clear
    TextBox2.Clear
    TextBox3.Clear
    Flag = FALSE
END
```

'This wipes the TextBoxes clean, for clarity after editing.
'Update the RowTextLabel information.
'Most users will consider the first row as Row 1.

If you only want to change data in one row/column location, you run the risk of writing from the other two (empty) TextBoxes (because you update from all three TextBoxes at once). Therefore, fill the TextBoxes with current data first, before you start editing. Since you single-click to move around the ColumnView, you would have distracting and constant changes of loaded data in the TextBoxes., if we allowed editing after single-click events. Therefore, it is safer and cleaner to limit the loading and updating of data to a double-click _Activate event. We use the Boolean “Flag” variable to signal this to the sub that handles updating.

```
PUBLIC SUB ColumnView1_Activate()
    'This loads the current data.
    sKey = ColumnView1.Current.Key
    TextLabel1.Text = CInt(sKey) + 1
    TextBox1.Text = ColumnView1.Current[0]
    TextBox2.Text = ColumnView1.Current[1]
```

```
TextBox3.Text = ColumnView1.Current[2]
Flag = TRUE
END

PUBLIC SUB Button4_Click() 'This allows you to update the data in the ColumnView.
IF NOT Flag THEN RETURN 'Prevents over-writing data from empty TextBoxes. No double-click (to load current data), no
editing!
sKey = ColumnView1.Current.Key
TextLabel1.Text = CInt(sKey) + 1
ColumnView1.Current[0] = TextBox1.Text
ColumnView1.Current[1] = TextBox2.Text
ColumnView1.Current[2] = TextBox3.Text
InitialTextBox
END

PUBLIC SUB ColumnView1_Click() 'This updates Row TextLabel, for fun,
sKey = ColumnView1.Current.Key      'when nothing else is going on.
TextLabel1.Text = CInt(sKey) + 1
END
```

ColumnView is very versatile – in fact, you can turn the first column into a TreeView!! For example, if you were constructing a genealogy program, you would use all of the same syntax as used with TreeView, for the items that will be inserted into the first column – except you would use “ColumnView1” instead of “TreeView1”:

```
ColumnView1.Add("Key", "Name", picGender, "Parent")
```

Successive columns do not accept TreeView syntax, so you would insert data in those with the [Row][Column] syntax we have just studied for ColumnView (but not using Column 0 – that's now occupied by a TreeView). For example, Row0/Column1 could hold birth date, Row0/Column1 could hold birth city, etc. Of course, you would use the appropriate column headings. Have fun!

Layout Controls – HBox, VBox, HPanel and Vpanel

Layout properties determine the type of positioning components may take on that panel. Some Layouts allow more exact positioning than others, but each has its uses. The box layouts lay out their children one after the other. The horizontal box layout lays out widgets in a horizontal row, from left to right, while the vertical box layout lays out widgets in a vertical column, from top to bottom. Each panel is given a Layout property. Layout properties can be applied to all panels.

HBox and VBox

Both of these classes are containers that arrange their children either horizontally or vertically. Like all other toolbox controls, they inherit their attributes from the Container class. Both of these controls act like a Panel without a border whose Arrangement property would be set to *Arrange.Horizontal*. These classes are creatable. Standard calling convention to create a new Hbox is:

```
DIM hHBox AS HBox  
DIM hVBox AS VBox  
hHBox = NEW HBox ( Parent AS Container )  
hVBox = NEW VBox ( Parent AS Container )
```

HPanel and Vpanel

Both of these classes are containers that arranges their children from top to bottom, and then left to right for the Hpanel and left to right, top to bottom for the VPanel. They are both like an ordinary Panel control without a border but whose Arrangement property would be set to *Arrange.TopBottom* (where a container stacks its children from top to bottom; if the children controls can't fit vertically, a new column of controls is started just after the previous column) or *Arrange.LeftRight* (where a container stacks its children from left to right; if the children controls can't fit horizontally, a new row of controls is started below the previous row). Both these types of controls inherit their attributes from the Container class. This class is creatable. Standard calling convention is:

```
DIM hHPanel AS HPanel  
DIM hVPanel AS VPanel  
hHPanel = NEW HPanel ( Parent AS Container )  
hVPanel = NEW VPanel ( Parent AS Container )
```

Let's create a simple program that will demonstrate each of these layout controls and show how they can be used in an application. For this application, start Gambas and create a QT graphical user interface program. Name it Layouts and double-click on the FMain icon to open the startup class form, FMain.form. For this application, we will need to use some icons. Most Linux-based distributions have icon files stored in the folder named:

```
filesystem:/usr/share/icons/default-kde/32x32/actions
```

Your system may be different. Regardless of where you find them, pick out seven icons and copy them to the Layouts folder. For our purposes, any seven will do but if you can find undo, redo, help, configure, colorize, printer, and exit type icons it will be easier. Here are the icons we chose to use for this project:

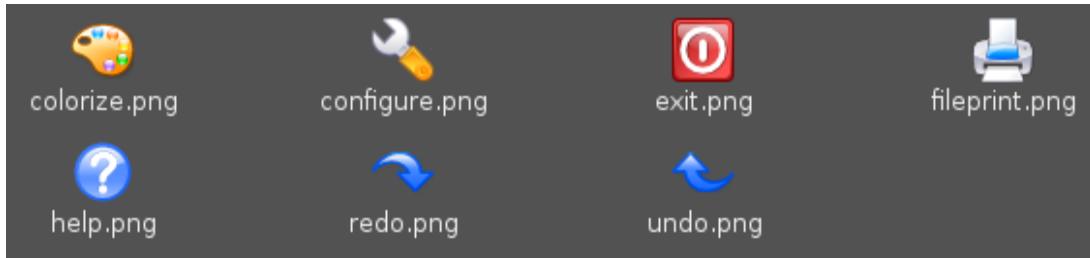


Figure 68: Layout project icons.

Our program is quite simple. We are going to create an HBox, a VBox, an HPanel and a VPanel and put some simple controls in each container. We will simply update a text label to show when something has been clicked. Our form will look like the one shown in following figure (in design mode).

From the top left of the figure above, we start with a VBox and a TextLabel. Below the TextLabel is the HBox. When you look for the VBox, HBox, HPanel and VPanel controls in the Gambas Toolbox, you will find them in the “Container” tab. When you create these controls, GB will suggest default names for each control. Take the default names for each control because our program uses those defaults. Place three ToolButtons in the VBox and four ToolButtons in the HBox.

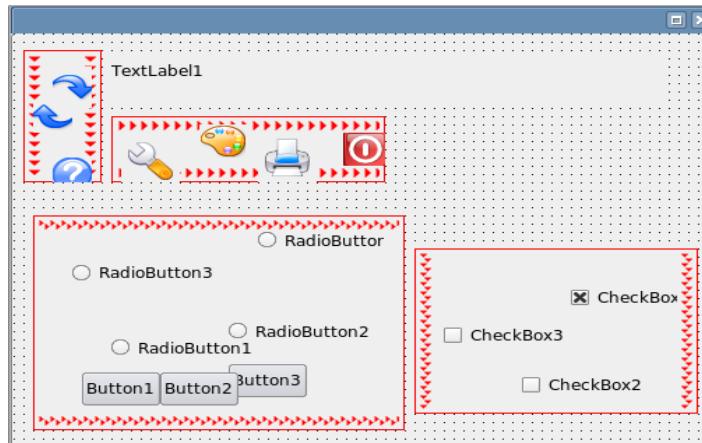


Figure 69: Form1 design mode showing layout of our controls.

Next, set the picture properties of each ToolButton to the icons you chose to use (hopefully the same as those above) for our project. Once you have the icons displayed, we need to start with the first ToolButton in the VBox and double-click to bring up the code window. You will need to do this for each of the seven icons and add code as follows:

' Gambas class file

```
PUBLIC SUB ToolButton1_Click()
    TextLabel1.Text = "Vert Undo clicked."
END
```

```
PUBLIC SUB ToolButton2_Click()
    TextLabel1.Text = "Vert Redo clicked."
END

PUBLIC SUB ToolButton3_Click()
    TextLabel1.Text = "Vert Help clicked."
END

PUBLIC SUB ToolButton4_Click()
    TextLabel1.Text = "Horiz Configure Btn clicked."
END

PUBLIC SUB ToolButton5_Click()
    TextLabel1.Text = "Horiz Color Button clicked."
END

PUBLIC SUB ToolButton6_Click()
    TextLabel1.Text = "Horiz Printer Btn clicked."
END

PUBLIC SUB ToolButton7_Click()
    TextLabel1.Text = "Horiz Exit Button clicked."
    WAIT 0.5
    ME.Close
END
```

The next control container we will create is the HPanel, as shown in the figure above. We will add four radiobuttons and three regular buttons to this container control. Try to arrange the buttons as shown in the figure. You will see why this is important once you run the program and see how the HPanel rearranges the controls. Once again, double click on each radio button and add the following code:

```
PUBLIC SUB RadioButton1_Click()
    TextLabel1.Text = "RadioButton 1 clicked."
END

PUBLIC SUB RadioButton2_Click()
    TextLabel1.Text = "RadioButton 2 clicked."
END

PUBLIC SUB RadioButton3_Click()
    TextLabel1.Text = "RadioButton 3 clicked."
```

END

```
PUBLIC SUB RadioButton4_Click()
    TextLabel1.Text = "RadioButton 4 clicked."
END
```

```
PUBLIC SUB Button1_Click()
    TextLabel1.Text = "Button 1 clicked."
END
```

```
PUBLIC SUB Button2_Click()
    TextLabel1.Text = "Button 2 clicked."
END
```

```
PUBLIC SUB Button3_Click()
    TextLabel1.Text = "Button 3 clicked."
END
```

The last thing we will do is create the VPanel with three checkboxes. Remember to arrange the checkboxes in your form like those in the picture above to see how the control rearranges them automatically. Add this code when you have placed the checkboxes in the VPanel:

```
PUBLIC SUB CheckBox1_Click()
    IF CheckBox1.Value = TRUE THEN
        TextLabel1.Text = "Checkbox 1 checked."
    ELSE
        TextLabel1.Text = "Checkbox 1 unchecked."
    ENDIF
END
```

```
PUBLIC SUB CheckBox2_Click()
    IF CheckBox2.Value = TRUE THEN
        TextLabel1.Text = "Checkbox 2 checked."
    ELSE
        TextLabel1.Text = "Checkbox 2 unchecked."
    ENDIF
END
```

```
PUBLIC SUB CheckBox3_Click()
    IF CheckBox3.Value = TRUE THEN
        TextLabel1.Text = "Checkbox 3 checked."
    ELSE
```

```
TextLabel1.Text = "Checkbox 3 unchecked."  
ENDIF  
END
```

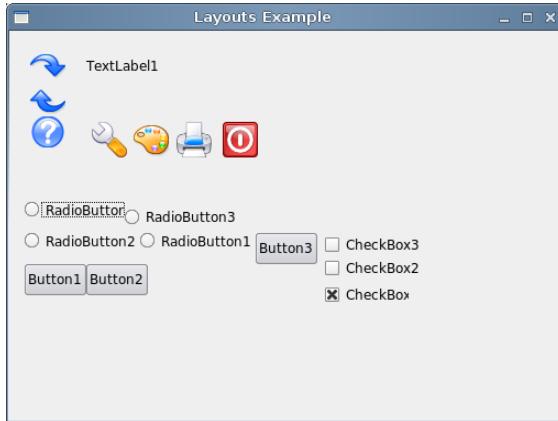


Figure 70: Layout program at program startup.

Now save your work and execute this simple program. You should see something similar to the figure on the right. Notice how the radiobuttons in the HPanel have aligned vertically from RadioButton4 to RadioButton1 and, if there was room it placed the button controls to the right and continued the alignment from top to bottom, left to right. For the VPanel, see how the checkboxes aligned themselves. These types of controls are unpredictable in how they will layout their children. It is best to use these in code dynamically when you have to create forms on the fly and do not have the opportunity to lay them out in a fixed panel in design mode.

The VBox and HBox controls hold the ToolButtons and allow you to create nifty little toolbars that work well with iconic driven controls, such as those depicted with our icons. Once you have finished playing around with our little application, close the program and we will move on to learn about TabStrips.

The TabStrip Control

The GB TabStrip control implements a tabbed container control. Like all controls, it inherits its attributes from the Container class. This class is creatable. This class acts like a read-only array. The standard calling convention is:

```
DIM hTabStrip AS TabStrip  
hTabStrip = NEW TabStrip ( Parent AS Container )
```

```
DIM hTab AS .Tab
```

hTab = hTabStrip [Index AS Integer]

Invoking the code above will return a virtual tab object from its index. TabStrips are useful for organizing and presenting information in self-contained units. TabStrips allow you to obtain input from users by presenting a simple, easy to follow interface that can literally guide them through a configuration or setup process. Here is a sample of a TabStrip in use:

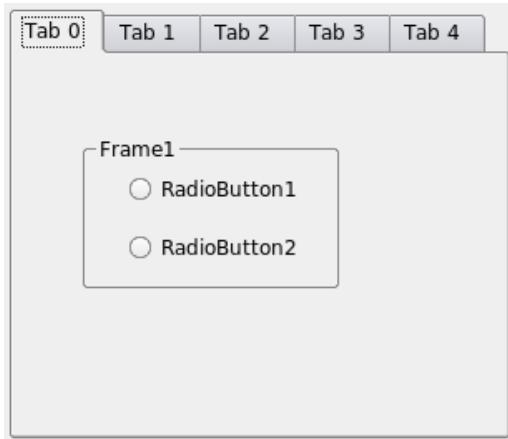


Figure 71: A TabStrip control.

To demonstrate how easy TabStrips are to create, we will build a small application that implements the tabbed interface you see above. Our application will show you how to use tabbed controls to present information to a user and respond to it in your applications. Start Gambas and create a new Qt graphical user interface program.

Name the project Tabs, click through the project creation wizard and when you get to the GB IDE, double-click on FMain. Next, you will need to place a TabStrip control on the new form. Once you place the TabStrip, you will only see one tab displayed. You must go to the properties window and set the Count property to 5 to have the control look like the one we show above. Note that tab numbering starts at zero in Gambas. Each tab acts like a container for the controls you place on it.

We will also need to add aTextLabel and a button to our form. TheTextLabel will be used to show the user actions in response to our tabbed interface. The button will be a Quit button which we will use to terminate the application. You will also need to copy the configure, help, and exit icons from the Layout project to the Tabs folder so we can use them with this project. GB is designed to contain all project work from the IDE to the folder you create when you start a project. As a result, you need to move out of the IDE to copy files from another folder to the current project (a quirk by design, perhaps?).

Anyway, let's get going on this project. Here is what your form should look like as you start to design the interface:

A Beginner's Guide to Gambas – Revised Edition

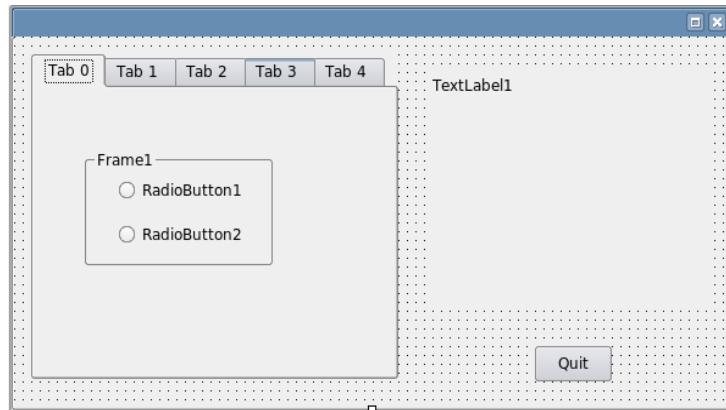


Figure 72: Tabbed form design layout.

For Tab0 we will add a frame and place two radiobuttons in it, like this:

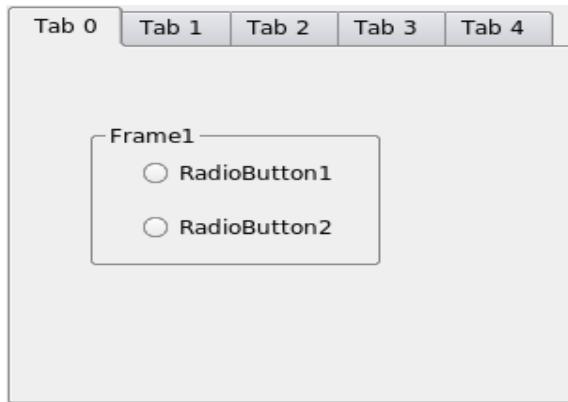


Figure 73: Tab0 layout.

Next, double click on each RadioButton and enter this code:

```
PUBLIC SUB RadioButton1_Click()
TextLabel1.Text = "You have clicked:<br><center><strong>RadioButton1</strong></center>"
END

PUBLIC SUB RadioButton2_Click()
TextLabel1.Text = "You have clicked:<br><center><strong>RadioButton2</strong></center>"
END
```

Here is what we will do for Tab1:

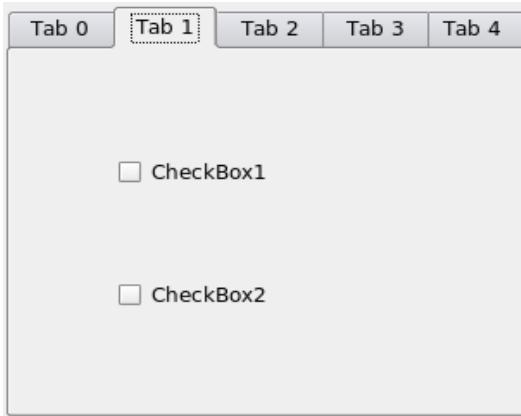


Figure 74: Tab1 layout.

Double-click on each checkbox and enter this code:

```
PUBLIC SUB CheckBox1_Click()
IF CheckBox1.Value = TRUE THEN
    TextLabel1.Text = "You have checked:<br><center><strong>CheckBox1"
ELSE
    TextLabel1.Text = "You have unchecked:<br><center><strong>CheckBox1"
ENDIF
END

PUBLIC SUB CheckBox2_Click()
IF CheckBox2.Value = TRUE THEN
    TextLabel1.Text = "You have checked:<br><center><strong>CheckBox2"
ELSE
    TextLabel1.Text = "You have unchecked:<br><center><strong>CheckBox2"
ENDIF
END
```

The code above will determine whether the user is checking an item or unchecking it by first looking at the Value property. If it is TRUE, then the box is already checked and we will indicate that fact in the update of the TextLabel1.Text. Otherwise, it is being unchecked and we will update the label to state that fact.

For Tab2, we are going to place a Panel control on the Tab and add three ToolButtons. Do not make the mistake of placing one ToolButton and trying to copy it and move the copy to the panel. It will not be recognized as a child of the panel control. It will appear on all tabs (*I believe this is another quirk of Gambas*).

For the Panel to accept the ToolButtons as children, you must individually click the ToolButton on the ToolBox and create each one just as you did the first ToolButton. For each ToolButton, assign the picture property the name of the icon we are using for that button, as

shown here:

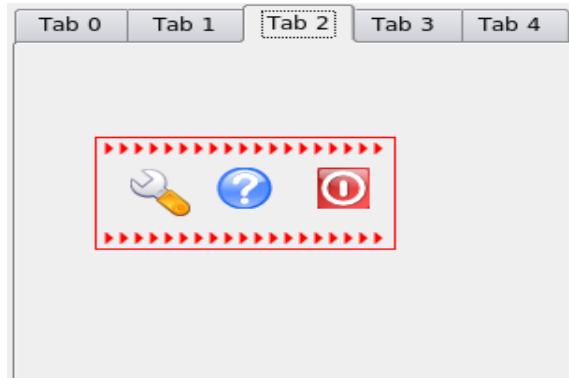


Figure 75: Tab2 ToolButton layout with icons.

The next step is to double-click on each ToolButton and add this code:

```
PUBLIC SUB ToolButton1_Click()
TextLabel1.Text = "You have clicked:<br><center>the <strong>Configure Icon</strong>"
END
```

```
PUBLIC SUB ToolButton2_Click()
TextLabel1.Text = "You have clicked:<br><center>the <strong>Help Icon</strong>"
END
```

```
PUBLIC SUB ToolButton3_Click()
TextLabel1.Text = "You have clicked:<br><center>the <strong>Exit Icon</strong>"
END
```

Now, we are going to add a ComboBox to Tab3. Nothing fancy, just a simple ComboBox with three items. Select the ComboBox control and place it on the tab as shown below:

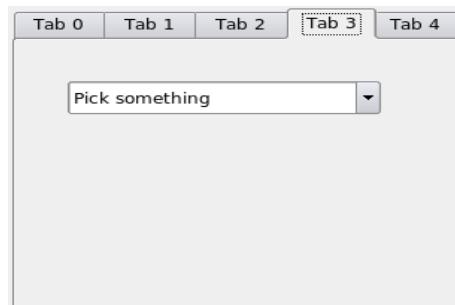


Figure 76: Tab3 layout with a ComboBox.

Remember, in order to set the items in the ComboBox, you need to go to the Properties window and select the List property. A button with three dots will appear (indicating that it leads to the List Editor) and you need to click that. Add the following items: “Pick something” (to display as our default text), “This thing”, “That thing”, and “Some other thing” to the list. Finally, we need to set an event for the ComboBox by single-clicking on the ComboBox control then right-clicking the mouse. Choose *Events* and select the *Change* event. Anytime the ComboBox changes, we want to display the changed text on our TextLabel. Now, add this code in the Code Editor for the ComboBox1_Change() event:

```
PUBLIC SUB ComboBox1_Change()
DIM comboitem AS String
comboitem = ComboBox1.Text
TextLabel1.Text = "You picked item:<br><center><strong> " & comboitem
END
```

For our last tab, Tab4, we are going to simply display the current time. This will require use of the Timer control. We are going to need to add a Label and a Timer control to Tab4, as shown below. The Timer tool can be found in the Special tab of the Gambas toolbox.

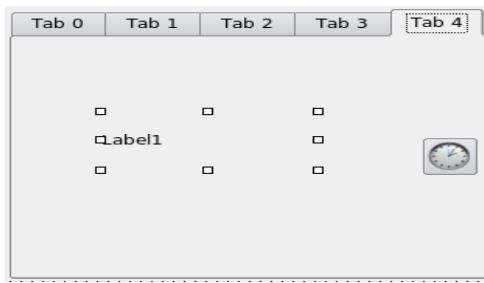


Figure 77: Tab4 layout with a Timer.

In order for the timer to work, we must enable it when the form first starts. Double-click on the form somewhere except on a control and you will see the Form_Open() subroutine appear in the code window. Add this code to the routine:

```
PUBLIC SUB Form_Open()
FMain.Caption = " Playing with TabStrips "
Timer1.Enabled = TRUE
TextLabel1.Text = ""
END
```

To actually use the timer, we need to set an Event. We do this by single-clicking on the Timer control then right-clicking the mouse. Choose *Events* and select the *Timer* event. You will be placed in the Code Editor and you need to add a single line of code:

```
PUBLIC SUB Timer1_Timer()
Label1.Text = Str$(Now)
END
```

This line of code will update the Label1.Text field every second that the Tab4 is displayed. In effect, you have created a clock on the Tab by doing this. That is all there is to it. Finally, double-click on your Quit button, and add this bit of code:

```
PUBLIC SUB Button1_Click()
    ME.Close
END
```

Run the program and see that it works. Next, we move to file operations in Chapter 9.



Chapter 9 – Working with Files

We are going to introduce you to the Gambas file management and file input/output (*file i/o*) operations in this chapter. Gambas has a full-featured set of functions and subroutines to support nearly any file i/o operation you may need to implement in your programs.

- ✓ Access
- ✓ Dir
- ✓ Eof
- ✓ Exist
- ✓ IsDir
- ✓ Lof
- ✓ Stat
- ✓ Temp / Temp\$
- ✓ OPEN and CLOSE
- ✓ INPUT, LINE INPUT, and PRINT
- ✓ READ, SEEK, WRITE and FLUSH
- ✓ COPY, KILL and RENAME
- ✓ MKDIR and RMDIR
- ✓ LINK

Access

Access is a function used to determine if a file is accessible or not. The calling convention is

Accessible = Access (Path [, Mode])

The Access function call returns TRUE if the file specified by Path is accessible. If the value of Mode is **gb.Read**, then the function returns TRUE if the file can be read. This is the default value for all files. If Mode is **gb.Write**, then Access returns TRUE if the file can be written to. When **gb.Exec** is specified for Mode, the function returns TRUE if the file can be executed. The previous flags can be combined with the OR operator. For a directory, the execution flag means that the directory can be browsed. For example:

PRINT Access("/home/benoit", gb.Write OR gb.Exec)

would return

True

while the statement

PRINT Access("/root", gb.Write)

will return

False

Dir

Dir returns a string array that contains the names of files located in *Directory* that match the specified File pattern. If no pattern is specified, any file name that exists in the directory is returned. The pattern may contain the same generic characters permitted for the LIKE operator. In other words, those characters found in the following table:

Generic character	Matches
*	Any number of any character or string.
?	Placeholder to specify the string position(s) for the searched-for character(s)
[abc]	Any character between the brackets.
[x-y]	Any character in the interval.
[^x-y]	Any character not in the interval.

The special generic character \ prevents the character following it from being interpreted as a generic. Standard calling convention is:

File name array = Dir (Directory [, File pattern])

Here is an example code segment to help you

' Print a directory

```
SUB PrintDirectory(Directory AS String)
    DIM sFile AS String
```

```
FOR EACH sFile IN Dir(Directory, "*.*")
    PRINT sFile
NEXT
END
```

Eof

The Eof function returns a Boolean value of TRUE if we are at the end of the stream. Standard calling convention is:

Boolean = Eof (File)

An example of how Eof is used follows:

```
...
OPEN FileName FOR READ AS #hFile
WHILE NOT Eof(hFile)
    LINE INPUT #hFile, OneLine
    PRINT OneLine
WEND
CLOSE #hFile
...
```

Exist

The calling convention for Exist is:

Boolean = Exist (Path)

It returns TRUE if a file or a directory exists. **NOTE: Using ~ as directory identifier does not work.** If the path specified does not exist, FALSE is returned. Here is an example of using Exist to determine if a file exists before calling OPEN to read the file:

```
...
DIM sCurrentFileName AS String
sCurrentFileName = "The Raven.txt"
IF Exist(sCurrentFileName) THEN
    ... 'open the file
ELSE
    Message.Info("File " & sCurrentFileName & " does not exist.")
ENDIF
...
```

IsDir

The function IsDir returns TRUE if a path points to a directory. If the path does not exist or is not a directory, this function returns FALSE. The calling convention is:

Boolean = IsDir (Path)

Some examples of how to use IsDir, as provided on the Gambas Wiki follow:

```
PRINT IsDir("/etc/password")
False
```

```
PRINT IsDir(Application.Home & ".gambas")
True
```

```
PRINT IsDir("/windows")
False
```

Stat

The Stat function returns information about a file or a directory. The File object information returned includes the type, size, last modification date/time, permissions, etc. The following file properties can be obtained using this function:

.Group	.Hidden	.LastAccess	.LastChange
.LastModified	.Link	.Mode	.Path
.Perm	.SetGID	.SetUID	.Size
.Sticky	.Time	.Type	.User

The standard calling convention for Stat is:

File info = Stat (Path)

Here is an example of how to use Stat and what will be returned from the console:

```
WITH Stat("/home")
PRINT .Type = gb.Directory
PRINT Round(.Size / 1024); "K"
END WITH
```

The console will respond with:

True
4K

Let's divert our attention for a moment to examine the syntax above. As you'll recall from Chapter 2, the “**WITH..END WITH**” statement is mainly used to assign property values. An expression beginning with a dot (“.”) located between **WITH** and **END WITH** refers to the object named after **WITH** – in this case, the “/home” subdirectory. “**gb**” is a static class that includes all of the basic Gambas constants which are used by many functions. “**gb.Directory**” is the flag constant used by the **Stat.Type** property to designate a directory. When the **Stat** function was invoked with a file path argument, the **Stat.Type** value provided was compared to the **gb.Directory** flag. Since **Stat.Type** cannot be assigned a value different from what the system reports, the equivalence in this case is not the result of an assignment. Gambas treats it as a Boolean test of whether or not this specific path, as reported by the system, had the **gb.Directory** flag set to TRUE. **Stat.Size** returns the size of the file or directory. In order to PRINT the directory size in kilobyte form, the value (in bytes) returned by **Stat.Size** was divided by 1024 bytes/kilobyte, then rounded to the nearest integer with the Round function, which we will encounter in Chapter 10.

Temp / Temp\$

Temp\$ returns a unique file name useful to create temporary files. The filename created will be located in the /tmp directory. Calling convention is:

File name = Temp\$([Name])

The path that is returned has the form:

/tmp/gambas.[User ID]/[Process ID]/[Name].tmp

If no optional filename prefix is included, then Gambas will assign an integer instead, and successive integers for additional temporary files.

Examples:

PRINT Temp\$()

returns:

/tmp/gambas.0/12555/1.tmp

PRINT Temp\$(another)

returns

/tmp/gambas.0/12555/another.tmp

OPEN and CLOSE

OPEN and **CLOSE** work as a team. **OPEN** opens a file for reading, writing, creating or appending data. Unless the **CREATE** keyword is specified, the file must exist. If the **CREATE** keyword is specified, then the file is created, or cleared if it already exists. If the file is opened for writing, the path to it must be absolute because relative paths are, by default, assumed to refer to files that exist inside the current Gambas project or archive folder. Calling convention for **OPEN** is:

file = OPEN path/filename FOR [READ|INPUT] [WRITE|OUTPUT] [CREATE | APPEND] [WATCH]

If the **APPEND** keyword is specified, then the file pointer is moved to the end of file just after the file is opened. If you use **INPUT** or **OUTPUT**, then the input-output are buffered. If you use the **READ** or **WRITE** keywords, the input-output are not buffered. If the **WATCH** keyword is specified, the file is watched by the interpreter using the following conditional guidelines:

- If at least one byte can be read from the file, then the event handler **File_Read()** is called.
- If at least one byte can be written into the file, then the event handler **File_Write()** is called.

Note about “Endianness” and “Streams”: For our purposes, “Endianness” refers to the byte order of bytes stored in a data file. If the most significant byte is stored in first position, followed by decreasingly significant bytes, then the format is “big-endian”. An example of the big-endian concept would be the number “twenty-four”. In “little-endian” format, the least significant byte is encountered first. The same number, expressed in a little-endian format, would be “four and twenty”. Endianness is important when you want to read data from a “stream” as binary data. “Stream” refers to access of any file from a process, whether it’s a finite stream like a file on disk, or an infinite stream like data from a videocam or data transmitted over the Internet via TCP (Transmission Control Protocol).

The following examples shows how to read data from a BMP file, known to use little-endian format:

```
DIM hFile AS File  
DIM iData AS Integer
```

```
hFile = OPEN "image.bmp" FOR INPUT  
hFile.ByteOrder = gb.LittleEndian
```

...
READ #hFile, iData

CLOSE is the reciprocal function and simply closes an opened file. Syntax for using **CLOSE** is:

CLOSE # File

Here is an example that opens and closes a file:

```
' Gambas Class File

sCurrentFileName AS String
sCurrentFileLength AS Integer

PUBLIC SUB Form_Open()

DIM sInputLine AS String
DIM hFileIn AS File

sCurrentFileName = "The Raven.txt"
IF Exist(sCurrentFileName) THEN
    hFileIn = OPEN sCurrentFileName FOR READ
    sCurrentFileLength = Lof(hFileIn)
    WHILE NOT Eof(hFileIn)
        LINE INPUT #hFileIn, sInputLine 'explained below
        TextArea1.Text= TextArea1.Text & Chr(13) & Chr(10) & sInputLine
    WEND
    CLOSE hFileIn
ELSE
    Message.Info("File " & sCurrentFileName & " does not exist.")
ENDIF
END
```

LINE INPUT

LINE INPUT reads an entire line of text on the standard input. It can be used on the console or with files. The calling convention is:

LINE INPUT Variable

or, for files where the data is read from the stream File:

LINE INPUT # File , Variable

Here is an example you can try to see how **LINE INPUT** works:

```
STATIC PUBLIC SUB Main()
  DIM hFile AS File
  DIM sInputLine AS String
  DIM lineCtr AS Integer

  hFile = OPEN "..\The Raven.txt" FOR READ
  WHILE NOT Eof(hFile)
    LINE INPUT #hFile, sInputLine
    PRINT Str(lineCtr) & Chr(9) & sInputLine
    INC lineCtr
  WEND
  CLOSE #hFile
END
```

NOTE: You should not use **LINE INPUT** call to read data from a binary file because it will lose the linefeed characters. For reading data from binary files, use **READ** instead:

READ #hFile, OneLine, Length

where **Length** is the maximum number of bytes you want read.. The **Length** can be any number, but you should limit it to the maximum size of any variable your program will use to hold the read data.

READ, SEEK, WRITE and FLUSH

The **READ** function reads data from the standard output. It treats the input stream as binary data whose data type is specified by the type of the variable the data is stored in during the read operation. The binary representation of the data should be the same representation created by use of the **WRITE** instruction. If *Variable* is a string, you can specify a length that indicates the number of bytes to read. If the value of the *Length* parameter is negative, then *-Length* bytes are read from the end of stream. If no *Length* parameter is specified for a string, the entire string is read from the stream. The string value then must have been written with the **WRITE** instruction. Standard calling convention for **READ** is:

READ Variable [, Length]

If **READ** is used with file operations, the calling convention is:

READ # File , Variable [, Length]

The **WRITE** function writes expressions to the standard output using their binary representation. If Expression is a string, you can specify a length that indicates the number of bytes to write. If no length is specified for a string value, the entire string value is written directly to the stream. Here is the calling convention for writing to a standard output stream (i.e., the console):

WRITE Expression [, Length]

and for writing data to a file, this is the convention:

WRITE [# File ,] Expression [, Length]

or

WRITE [# File ,] Expression AS Datatype

The **SEEK** function defines (or moves) the position of the stream pointer, most often in preparation for the next read/write operation. If the position specified by the *Length* parameter is negative, then the stream pointer is moved backwards *-Length* bytes from the end of the file. To move the stream pointer beyond the end of the file, you must use the Lof() function. Standard calling convention is:

SEEK # File , Position

Here are some examples of using **SEEK**:

' Move to the beginning of the file

SEEK #hFile, 0

' Move after the end of the file

SEEK #hFile, Lof(#hFile)

' Move 100 bytes before the end of the file

SEEK #hFile, -100

The **FLUSH** function is called to empty (or flush) the contents of a buffered data stream. If no stream is specified, every open stream is flushed. Typically, **FLUSH** is called after multiple write operations and before closing out a program or routine to ensure all data held in a buffer is written out to the file before exiting. Here is the calling convention for **FLUSH**:

FLUSH [[#] File]

COPY, KILL and RENAME

The **COPY** function will copy a source file to a specified destination file. The destination path need not have the same name than the source path. You cannot copy directories recursively with this function. The calling convention for this function is:

COPY Source path TO Destination path

Here is an example of how to use **COPY**:

```
' Save the gambas configuration file  
COPY User.Home &/.gambas/gambas.conf TO "/mnt/save/gambas.conf.save"
```

The **KILL** function removes a file. The calling convention is **KILL Path** and whatever exists in Path is permanently deleted. **MOVE** moves a file or folder to another location, or changes the name of a file or folder from *Old name* to *New name*. Here are examples that do both things – the first line changes the name in the current location. The second line moves the file to a different folder:

```
MOVE User.Home &/"The Ravenous.txt" TO User.Home &/"The Raven.txt"
```

```
MOVE User.Home &/"The Raven.txt" TO User.Home &/"Public/The Raven.txt"
```

MKDIR, RMDIR

The **MKDIR** function is used to create (make) a folder on your file system and **RMDIR** is used to remove a folder. Both functions take a single parameter, a File path and name. The calling convention is:

MKDIR User.Home &/"Documents/Gambas/test"

and

RMDIR User.Home &/"Documents/Gambas/test"

In order to demonstrate most of the file and I/O functions described in this chapter, we are going to create an application that will act as a simple text editor. The intent here is not to create a full-blown text editor, but to show you how to use these functions in the context of an application.

The application we will build in this chapter is a little more complex than what we have seen so far in that we will be creating two forms, an input file, a help file, and we will be using quite a few system icons which will be copied from the Linux OS distribution's system Icon folder(s). Here is what our application will look like at runtime:

A Beginner's Guide to Gambas – Revised Edition

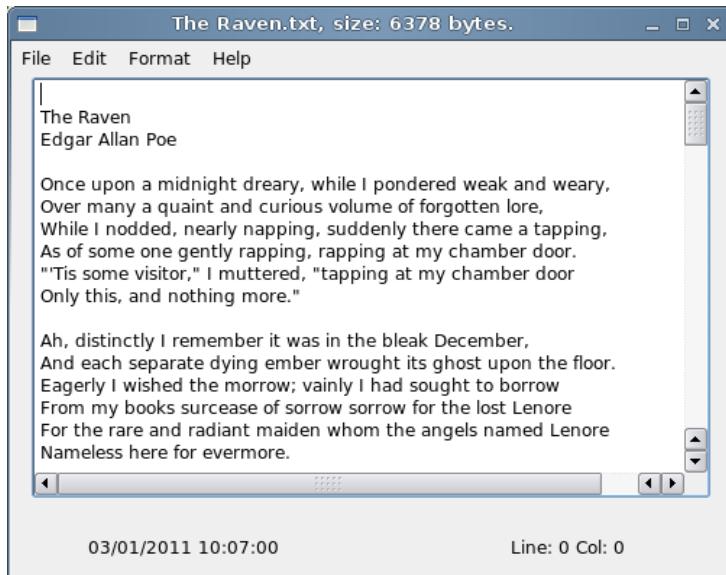


Figure 78: The *FileOpsQT* program at runtime.

In this program, we will create a main menu, a text area to display and edit text with, and we will create two `TextLabel` status elements at the bottom of the display. We will also introduce use of the Timer control (shown as the clock in the picture below). This form, named `FMain.form`, will look like this in design mode:

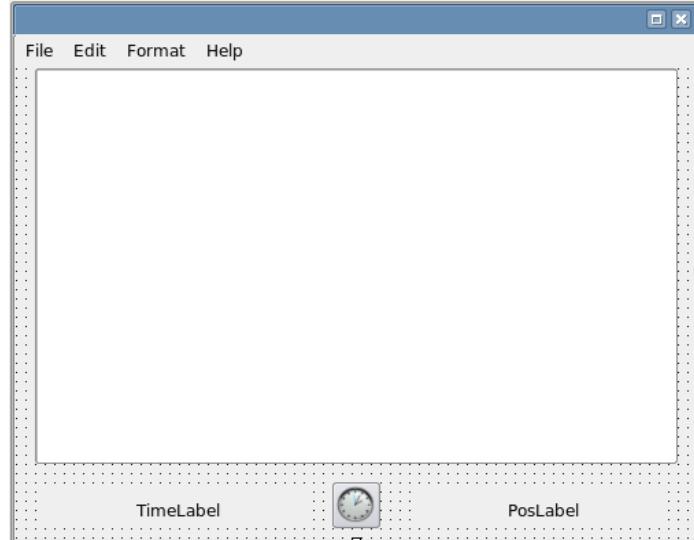


Figure 79: The sample application in design mode.

To start, load Gambas and create a new QT graphical user interface project. Name it `FileOpsQT` and when you get to the GB IDE, create a new form, `FMain`, a startup class. When the form displays on your screen, we are going to begin by creating the menus. You can press **CTRL-E** to bring up the Menu Editor. Add the following items using the editor. For the icons, you will need to copy icons from the same directory we specified in the last chapter.

A Beginner's Guide to Gambas – Revised Edition

Most Linux-based distributions have icon files stored in the folders named:

</home/usr/share/icons/default-kde/32x32/actions>
or
</home/usr/share/icons/default-kde/32x32/applications>

Your system may be different. Regardless of where you find them, pick out icons appropriate for our program and copy them to the FileOps folder. Here are the 17 icons chosen to use for this project:

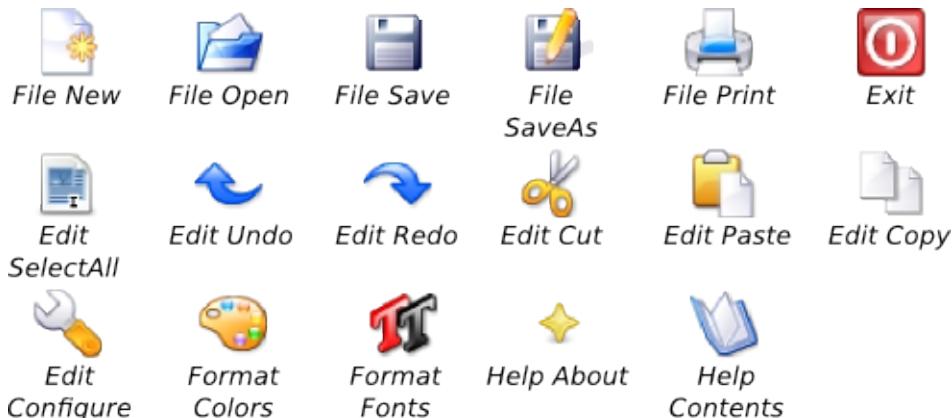


Figure 80: Sample application icons.

If you cannot find the exact same icon, simply pick one that is most appropriate and save it with the file name for that option as shown in the tables below. It is not crucial that every icon be exactly the same. In fact, the SelectAll icon and HelpAbout icon, which aren't found in the default.kde folder, can be created using the Gambas Icon Editor, or copied from the Gnome icons folder. Construct the file menu using the following tables. Remember to associate a "picture" with each menu item that is below the top level menu items.

FileMenu

Variable Name	Caption	Icon Property (filename)
FileNewItem	"New"	Picture["filenew.png"]
FileOpenItem	"Open..."	Picture["fileopen.png"]
FileSaveItem	"Save..."	Picture["filesave.png"]
FileSaveAsItem	"Save as..."	Picture["filesaveas.png"]
FilePrintSelect	"Print"	Picture["fileprint.png"]
FilePrintItem	Print (Gambas method)	(none)
FileSysPrintItem	Print (System method)	(none)
FileExitItem	"Exit"	Picture["exit.png"]

EditMenu

Variable Name	Caption	Icon Property (filename)
EditSelectAllItem	"Select All"	Picture["selectall.png"]
EditUndoItem	"Undo"	Picture["undo.png"]
EditRedoItem	"Redo"	Picture["redo.png"]
EditCutItem	"Cut"	Picture["editcut.png"]
EditCopyItem	"Copy"	Picture["editcopy.png"]
EditPasteItem	"Paste"	Picture["editpaste.png"]
EditPrefsItem	"Preferences"	Picture["configure.png"]

FormatMenu

Variable Name	Caption	Icon Property (filename)
FormatColorItem	"Colors"	Picture["colorize.png"]
FormatFontItem	"Font"	Picture["fonts.png"]

HelpMenu

HelpContentsItem	"Contents"	Picture["contents.png"]
HelpAboutItem	"About"	Picture["help-about.png"]

Here is the code for we will use for FMain. We will go thru this line by line so everything is fully explained. We start our program by declaring two class global variables, sCurrentFileName and sCurrentFileLength.

```
' Gambas class file
sCurrentFileName AS String
sCurrentFileLength AS Integer
```

When the program starts, this next subroutine is called. It calls a constructor, _new() to initialize. The constructor is explained in the next section. When our program starts, we are going to have it automatically load our default text file, named "The Raven.txt". This file was copied off the Internet and you can place any text file in the FileOps project folder to use for this project. Just remember that the filename has to be changed in the code below if you choose not to use our default filename.

```
PUBLIC SUB Form_Open()
```

```
    DIM sInputLine AS String  
    DIM hFileIn AS File
```

We will assign our filename to the sCurrentFileName string variable and check to see if the file exists in the project folder. If it exists, then the program will open the file and get the initial file length. We will display the filename and size in the Form1.Caption property. While it is not customary to display the file size like this, it is done here simply to demonstrate the use of the Lof function.

```
sCurrentFileName = "The Raven.txt"  
IF Exist(sCurrentFileName) THEN  
    hFileIn = OPEN sCurrentFileName FOR READ  
    sCurrentFileLength = Lof(hFileIn) 'to show how to get file size  
    FMain.Caption = " " & sCurrentFileName & ", size: " & Str(sCurrentFileLength) & " bytes."
```

Now, we will enter a loop structure to read all the lines of the text file into our TextArea1.Text property. We will continue to loop until we have reached the end of the file. As we read each line into memory and assign it to the variable sInputLine, we must remember to add the CRLF (a Carriage Return) (Chr(13)) and a Line Feed (Chr(10)) to the end of the line so it will display properly in the control.

```
WHILE NOT Eof(hFileIn)  
    LINE INPUT #hFileIn, sInputLine  
    TextArea1.Text= TextArea1.Text & Chr(13) & Chr(10) & sInputLine  
WEND
```

As soon as we exit the loop, meaning we have reached the end of the file, we will close the file.

```
CLOSE hFileIn
```

If our check to see if the file exists failed, we would enter this **ELSE** block of code and display a message indicating that the file did not exist. Next, we call our FileOpenItem_Click() subroutine to have the user pick a file to open.

```
ELSE  
    Message.Info("File " & sCurrentFileName & " does not exist.")  
    FileOpenItem_Click() ' force a file to be picked if no default.  
ENDIF  
END
```

That is all there is to the Form_Open() subroutine. Here is the constructor for the form:

```
PUBLIC SUB _new()
    Timer1.Delay = 1000
    Timer1.Enabled = TRUE
END
```

The constructor simply sets the timer delay to 1 second (1000 ms) and sets the enabled property to true. The net effect here is that the timer will update itself every second. We will implement the timer control later in this code.

Let's now turn our attention to the File | New menu option. For a new file to be created by the user, we must first blank out the text in the TextArea1.Text property. We will prompt the user for a new file name and save the blank file to disk to create it. Whatever changes the user will make will be saved on exit or whenever the user chooses to save from the menu. The last thing we do in this subroutine is set the FMain.Caption property to the newly saved filename.

```
PUBLIC SUB FileNewItem_Click()
    TextArea1.Text = ""
    Dialog.Title = "Enter new file name..."
    Dialog.Filter = [ "Text files (*.txt)", "All files (*.*)"]
    Dialog.SaveFile
    Form1.Caption = Dialog.Path
END
```

The next routine is activated when the user clicks the File | Open menu item.

```
PUBLIC SUB FileOpenItem_Click()
```

This routine requires two local variables. hFileIn is the handle to the FILE object we will open and sInputLine is a string variable that we will use to read the text file into our TextArea1.Text property.

```
DIM hFileIn AS File
DIM sInputLine AS String
```

Just before the call to the standard dialog, we can set the window caption (or title) using the Dialog.Title property. We will also set a filter for the dialog to only look for files with a .txt extension. As a last resort, we will let the user choose to view all types of files by selecting the filter *.* from within the dialog.

```
Dialog.Title = "Choose a text file"
Dialog.Filter = ["Text files (*.txt)", "All files (*.*)"]
```

A Beginner's Guide to Gambas – Revised Edition

When the call to Dialog.OpenFile is made, remember that it will return a TRUE value if the user clicks on the Cancel button, otherwise, FALSE is returned. We will check for a TRUE value and return if that is what we get. Otherwise, it means the user selected a file and we will process it.

```
IF Dialog.OpenFile() THEN  
    RETURN  
ELSE
```

The filename of the file that was selected by the user is stored in the Dialog.Path property. We will assign that to the global variable sCurrentFileName so other parts of the program will have access to it. We will set the FMain.Caption property to the name of the newly opened file and clear the current TextArea1 display by invoking the Clear method.

```
sCurrentFileName = Dialog.Path  
FMain.Caption = " " & sCurrentFileName & "  
TextArea1.Clear
```

Next, we will open the file and read each line into memory using the LINE INPUT function and our string variable *sInputLine*. Once the data is stored in *sInputLine*, we will add it to the end of whatever is currently stored in the TextArea1.Text property (effectively concatenating onto the end of the current text value) and add our CRLF characters. Once we reach the end of the file, we will exit the loop and close the file.

```
OPEN sCurrentFileName FOR READ AS hFileIn  
WHILE NOT Eof(hFileIn)  
    LINE INPUT #hFileIn, sInputLine  
    TextArea1.Text = TextArea1.Text & Chr(13) & Chr(10) & sInputLine  
WEND  
CLOSE hFileIn  
ENDIF  
END
```

The File | Save menu item, when clicked, will invoke this subroutine. It works almost like the previous one except that we already know the name of the file held in the global variable sCurrentFileName. When we return from the standard dialog call, we update the global variable with whatever was returned – not caring if the file is the same name or not. Whatever it is or has become will be saved.

```
PUBLIC SUB FileSaveItem_Click()  
  
    Dialog.Title = "Save text file"  
    Dialog.Filter = [ "Text files (*.txt)", "All files (*.*)"  
    IF Dialog.SaveFile() THEN  
        RETURN
```

```

ELSE
    sCurrentFileName = Dialog.Path
    File.Save(sCurrentFileName, TextArea1.Text)
ENDIF
END

```

The next subroutine we will code is the FileSaveAs menu item. If the user wants to save the existing file under a different filename, this is the routine to handle that option. For this subroutine, it will work almost identically to the FileSave subroutine but we will need to check to see if the filename returned from the standard file dialog is actually different than the current filename. If it is different, then we will save the file under the new name. If the name is the same, we will pop up a message box and inform the user that no save action was taken because the names were identical.

```

PUBLIC SUB FileSaveAsItem_Click()
    DIM sTmpName AS String

```

We use the local string variable sTmpName to make our check. First of all, we need to obtain the full path to the application and concatenate the current filename to that, to get the fully qualified path name for our file. We use the Application class for that purpose, checking the Path property to get the name and path of the running application. Note the use of the &/ operator which is specifically used to concatenate filenames.

```
sTmpName = Application.Path &/ sCurrentFileName
```

Just as we did previously, we will set the title and filter properties, then call the standard file dialog.

```

Dialog.Title = "Save text file as..."
Dialog.Filter = ["Text files (*.txt)", "All files (*.*)"]

```

If the standard dialog call returns TRUE, the user has canceled out of the routine and we will simply return to the calling process (i.e., our standard event loop).

```

IF Dialog.SaveFile() THEN
    RETURN

```

If the filenames are the same, we will pop up the message box and return:

```

ELSE IF sTmpName = Dialog.Path THEN
    Message.Info("File not saved: name same as current file.", "OK")
    RETURN

```

Otherwise, we set the global variable to the filename returned from the standard file dialog and call the Save method, passing the filename and our text as parameters. We update

the caption of the window as the last thing we do before returning to the calling process.

```
ELSE
    sCurrentFileName = Dialog.Path
    File.Save(sCurrentFileName, TextArea1.Text)
    FMain.Caption = sCurrentFileName
ENDIF
END
```

Printing any type of item is pretty easy in Gambas. Our File | Print menu item will invoke this subroutine when clicked. We will declare our local variables first and then something else we **MUST** do before we actually try to print anything is call the Printer.Setup() subroutine. It is *mandatory* to call this function in GB.

```
PUBLIC SUB FilePrintItem_Click()
    DIM iLeftMargin AS Integer
    DIM iTopMargin AS Integer
    DIM iRightMargin AS Integer
    DIM iBottomMargin AS Integer

    DIM arsTxtToPrint AS String[]
    DIM iTxtPosY AS Integer
    DIM sTxtLine AS String

    DIM strMsg AS String
    DIM iPrinterAreaWidth AS Integer
    DIM iPrinterAreaHeight AS Integer
    DIM aString AS String
```

Note the next 3 lines of code. The call to the Printer.Setup method automatically opens the printer dialog control. This dialog closes by the user selecting either OK or Cancel. When it closes, this method returns TRUE if the user clicked on the Cancel button, and FALSE if the user clicked on the OK button. This allows you to detect the user's intent, and exit the print job if needed.

Also, before you call the print dialog, you can over-ride the setup dialog's default settings for properties like size and resolution (and others) so that you don't have to set this every time you print something. For example, A4 is the default size in the print dialog (and standard outside the United States), but in the U.S. most people will expect to print on Letter-sized (8.5X11) paper.

```
Printer.Size = "Letter"
```

NOTE: THIS WILL BE “Printer.Paper = Printer.Letter” IN GAMBAS 3.

'older printers do not seem to be set to DPI lower than 600 in GB

Printer.Resolution = 300

If you do not include the next line of code, printing will proceed without the printer dialog being called.

IF Printer.Setup() THEN RETURN

NOTE: THIS WILL BE "Printer.Configure()" IN GAMBAS 3.

A note about the printer dialog: After the user has made various choices, the dialog returns values for each property that it can collect information about. The dialog does nothing else – it's up to the programmer to read those parameters and write code to make use of those choices. For example, the dialog allows the user to print fewer than all of the pages. However, in our program, we have not written a routine to read that choice and set that value, the way we did above for paper size and resolution.

For a 300 dpi printer, a value of 300 would equal a one inch margin. Now, we set the margin values by using the Printer properties (or the ones we over-rode above) returned from the Printer.Setup call.

'set left margin to one inch, equal to the DPI of the printer
iLeftMargin = Printer.Resolution

'set top margin to one inch, equal to the DPI of the printer
iTopMargin = Printer.Resolution

'set right margin to one inch, equal to the DPI of the printer
iRightMargin = Printer.Resolution

'set bottom margin to one inch, equal to the DPI of the printer
iBottomMargin = Printer.Resolution

'Define the Printable area width
iPrinterAreaWidth = Printer.Width - iLeftMargin - iRightMargin

'Define the Printable area height
iPrinterAreaHeight = Printer.Height - iTopMargin - iBottomMargin

Build the output string for our Message.Info call and display that data to user:

strMsg = "Res: " & Str(Printer.Resolution) & ", Width: " & Str(Printer.Width) & ", Height: " & Str(Printer.Height)

'show a MessageBox with data obtained from the printer setup call

```
message.info(strMsg)
```

Next, we need to parse all of the content of the text in the TextArea1 control into a string array. It is very easy to do in GB and only takes a single line of code. To do this, we will use the Split function and set the delimiter parameter to the '\n' character which designates the end of the line of text:

```
arsTxtToPrint = Split(TextArea1.Text, "\n")
```

We are ready to print. To do so, you **MUST** call the Begin method and specify that you want the Printer object to be the output device used by the Draw method.

```
Draw.Begin(Printer) ' Initializes the draw routine
```

In our code below, we have called the Draw.Rect function to draw a border at one inch margins around our page.

```
'make a border around the page
```

```
Draw.Rect(iLeftMargin,iTopMargin,iPrinterAreaWidth,iPrinterAreaHeight)
```

```
'now we will loop through each line in the array and print it
```

```
FOR EACH sTxtLine IN arsTxtToPrint
```

```
'increment the vertical (Y) line position for each line printed
```

```
'Draw.TextHeight returns the height in dots (the unit of resolution
```

```
'in dpi) of every line in sTxtLine
```

```
iTxtPosY = iTxtPosY + Draw.TextHeight(sTxtLine)
```

```
'Now, test the line position ... if is > the bottom of the printer
```

```
'area then we need to reset it to zero and perform a new page
```

```
IF iTxtPosY >= (iPrinterAreaHeight) THEN
```

```
    iTxtPosY = 0 'reinit the Current Text Position Y
```

```
    Printer.NewPage
```

```
    'make a border around the new page
```

```
    Draw.Rect(iLeftMargin,iTopMargin,iPrinterAreaWidth,iPrinterAreaHeight)
```

```
ENDIF 'our test to see if line falls past bottom margin
```

Now, we make the call to Draw.Text to actually print the text. The first parameter Draw.Text takes is the string object *sTxtLine* representing our line of text to be printed from the array of string *arsTxtToPrint*. The next two parameters, *iLeftMargin* and (*iTopMargin+iTxtPosY*) are used to define the left margin and the current vertical Text Y position. Each line is incremented by the value (in dots) held in the property **Draw.TextHeight**. Optionally, you can specify right and bottom margins, and an alignment property. In our case below, we specified the right margin, defined by *iPrinterAreaWidth* (as

A Beginner's Guide to Gambas – Revised Edition

we calculated above) and omitted the bottom margin since it is maintained in the code of the **FOR LOOP** above. We also added 10 to the dot value of iLeftMargin, to give some space between the left side of the rectangle and the beginning of each line. Finally, we have specified the alignment of the text to be left-aligned.

```
'Now we print the text line at the new line position  
Draw.Text(sTxtLine,iLeftMargin+10,iTopMargin+iTxtPosY,iPrinterAreaWidth,,Align.Left)  
NEXT 'iteration of the FOR LOOP
```

To finish the print process, **you must remember to call Draw.End:**

```
Draw.End 'Then print final contents and eject last page  
END
```

Since our program only works with text files, there is an alternate, sometimes easier method of printing. You can use the system's built-in pr (print) command. It is invoked using the Gambas SHELL method. This call can allow you to print any text file from within your program directly to the default printer. To see how this method works, let's add a menu item to our program and try this. Go to the FMain.form and press Print (system method) to create a click event and add this code:

```
PUBLIC SUB FileSysPrintItem_Click()  
DIM aString AS String  
  
aString = "pr -h """ & sCurrentFileName & "\" & "-o5<\"" & Application.Path &/ sCurrentFileName & "\" | lpr"  
  
message.Info(aString,"Ok")  
SHELL aString  
END
```

It looks a lot more difficult than it really is. Let's go through the code. First, we declare a string variable, aString. Next we build the string that will be passed off to the system as a console command. Because we must ensure the string is built exactly as it would have to be typed from the console, we must ensure it contains quote marks embedded in it. The first part of the string:

"pr -h

will invoke the system command pr. The -h parameter specifies that we want to have a header printed on each page, in this case, it will be the current file name, held in the variable sCurrentFileName, which is global in scope.

The next part of the string is:

\" & sCurrentFileName & "\\"

In the first group of characters (\””), \” is a switch that turns off the second “ as demarking a text substring. When we pass the completed string to the SHELL command, the filename must be enclosed in literal quotation marks. Therefore, after the \” switch, the second “ is just a literal opening quotation mark for the filename. In the group (“\””) following the filename, the first “ is the closing literal quote mark for the filename, the following \” restores the demarking function of “ for text substrings, and the final “ is the marker for the end of the first substring. Note that the &s must be used inside the literal quotes to concatenate the quote marks to the filename. If the &s were used outside of the quotes, you would concatenate the word “sCurrentFileName”, rather than the string value contained in the global variable. The next part is:

& “ -o5 < \”” & Application.Path &/ sCurrentFileName & “\”

which concatenates the -o parameter (which specifies a margin offset) and a value of 5 spaces. The < character is used on the command line to designate that the input stream (the file to be read to the printer) comes from the filename that follows the < character. In this case, we need to delimit our filename with quotes again, so the \” switch turns off the normal quote demarking function, and the “ that follows the switch is the opening literal quote for the path+filename. Once again, the &s that join this path+filename string to the rest of the substring must be inside the literal quotes. In the final group (“\””), the first “ is the closing literal quote for the path+substring, and the following \” switch restores the demarking function of the quote marks. The last part of our command is:

| lpr”

The | character *pipes* the output to the line printer (lpr). The final “ terminates the completed string that will describe the desired process to the SHELL command. In all, it takes one variable declaration and three lines of code to print a file using this technique. Either one will work so choose the one that suits you best. Although the string that is passed to SHELL is dense with information, and the syntax and spacing are critical, people who are experienced in using Linux commands from a terminal will find it fairly straightforward.

If the user clicks on the File | Exit menu item we could simply close the program. However, we want to give the user a chance to save before exit so we will pop up a question dialog to ask if they want to save the file. If so, we will save it. If not, we simply exit the program.

```
PUBLIC SUB FileExitItem_Click()
DIM iResult AS Integer

iResult = Message.Question("Save your work?", "Yes", "No", "Cancel")
SELECT CASE iResult
CASE 1
    File.Save(sCurrentFileName, TextArea1.Text)'Save, then close.
```

```
CASE 2 'Proceed to close without saving.  
CASE 3  
    RETURN 'Terminate SELECT..END SELECT without save or close.  
CASE ELSE  
END SELECT  
ME.close  
END
```

Note that in the code above, Message.Question() tags the elements inside the parentheses with the keys: 0, 1, 2, and 3. The integer iResult holds the key of the choice made by the user -- 0 never being a choice, since 0 is the question. Message.Question holds a collection, not an array, and collection keys can be strings. Therefore, it works equally well to dimension iResult AS String, rather than AS Integer.

That's it for the File Menu. Now, the real power of Gambas will become obvious as we build the code for the Edit menu functions. From this menu, we want the user to be able to select all text, undo, redo, cut, paste, copy or call a configuration routine. For our purposes, the configuration routine will be pretty simple, but it should be enough to show you how to accomplish the task.

If the user clicks on the Edit | Select All menu item, it will select all the text in the TextArea. This is done by obtaining the length of the text using the TextArea1.Length property and passing that value to the TextArea1.Selection Method with a starting point of zero and an ending point Length characters from that start point, as shown in the code below.

```
PUBLIC SUB EditSelectAllItem_Click()  
DIM ilength AS Integer  
ilength = TextArea1.Length  
TextArea1.Select(0,ilength)  
END
```

The code above could also have been written in a more simplified manner without using the Integer variable *iLength*. It could have been reduced to a single line, as shown below:

```
PUBLIC SUB EditSelectAllItem_Click()  
TextArea1.Select(0, TextArea1.Length)  
END
```

Either method will work but the author's preference is to use the second method as it reflects the simplicity of Gambas when programming. This simplicity is seen in the use of the built-in Undo, Redo, Cut, Paste and Copy methods provided by GB for TextArea1.Text as well. Gambas provides these methods so providing this capability for your program users is a simple matter of adding a single line of code for each menu item's click event, as shown below:

```
PUBLIC SUB EditUndoItem_Click()
```

```
    TextArea1.Undo
```

```
END
```

```
PUBLIC SUB EditRedoItem_Click()
```

```
    TextArea1.Redo
```

```
END
```

```
PUBLIC SUB EditCutItem_Click()
```

```
    TextArea1.Cut
```

```
END
```

```
PUBLIC SUB EditPasteItem_Click()
```

```
    TextArea1.Paste
```

```
END
```

```
PUBLIC SUB EditCopyItem_Click()
```

```
    TextArea1.Copy
```

```
END
```

Note: since Gambas now has the default first form named “Fmain.form” (essentially, “form0”), succeeding forms have the names “Form1.form”, “Form2.form”, etc.

Our Edit | Preferences menu option will be used to show you how to bring up a second form in your program. We will save the data gathered from the Form1.form in a configuration file named FileOps.conf and that will enable us to pass those saved values back and forth between the main form, Form1.form and the second form, Form2.form.

This is the easiest way to exchange data between classes but it is possible to pass data as parameters between forms. We will discuss that later in this book. The EditPrefsItem_Click() subroutine is pretty simple. We just show the second form.

```
PUBLIC SUB EditPrefsItem_Click()
```

```
    Form1.Show
```

```
END
```

Here is what Form1.form will look like:

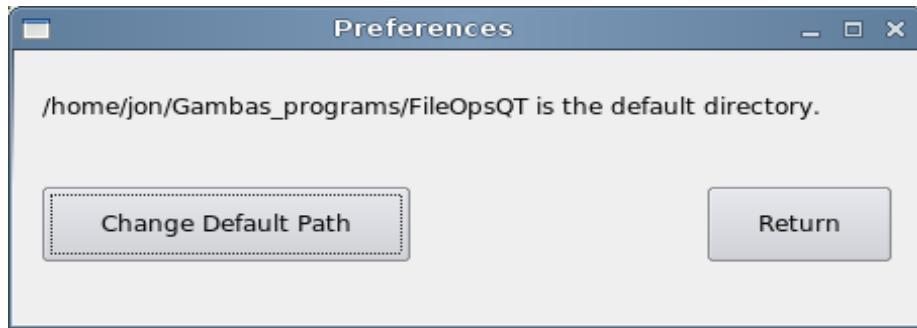


Figure 81: Form1.form design mode.

As you can see, it is pretty basic. A `TextLabel` is used to display the default path. There are two buttons, one to change the default path by calling the standard `Dialog.SelectDirectory` routine and the other to return to the calling process, `Fmain.form` in this case. Here is the code for our `Form1`:

```
' Gambas class file
```

When the form is opened, we will display the caption “Preferences” at the top of the window. The constructor routine `_new()` is called automatically.

```
PUBLIC SUB Form_Open()
    Form1.Caption = "Preferences"
END
```

The constructor will call our subroutine `ReadPrefs`, passing the name of the configuration file as a parameter. `FileOps.conf` is the name we will use for holding our application's config data. Next, we populate the `TextLabel1.Text` property with the current default directory, which defaults to where the application resides on disk.

```
PUBLIC SUB _new()
    ReadPrefs("FileOps.conf")
    TextLabel1.Text = Application.Path & " is the default directory."
END
```

When the user clicks the Change Default Path button, the following routine is called:

```
PUBLIC SUB Button1_Click()
    Dialog.SelectDirectory
    TextLabel1.Text = Dialog.Path
END
```

The subroutine above simply calls the Gambas standard dialog function `Dialog.SelectDirectory` and updates the `TextLabel1.Text` property with what the user has

chosen as the default value. When the user returns from this subroutine, the only other option on the form is to click the Return button, which executes this code:

```
PUBLIC SUB Button2_Click()
  DIM MyConfig AS String

  MyConfig = "FileOps.conf"
  WritePrefs(MyConfig)
  Form1.Close
END
```

As you can see above, a local string variable MyConfig is declared and set to the configuration filename FileOps.conf. Next, we call our WritePrefs subroutine and pass the value of MyConfig as a parameter. That filename will be concatenated with the newly selected path that the user sets in Dialog.Path, will be stored in the file and can be read by any other class that needs to find out this information. Note, however, that this exercise will not result in your new preference being read from that configuration file – we are still reading the default preference from the FileOps.conf file located in our project folder, when we call the ReadPrefs() subroutine. A workaround would be to create a new file in the project folder to hold the location of the configuration file, but chasing our configuration file around is not our purpose. We merely want to show the use of the Dialog.Path control.

Here is the ReadPrefs() subroutine for accomplishing this task:

```
PUBLIC SUB ReadPrefs( filename AS String )
  'declare our local variables
  DIM sInputLine AS String
  DIM hFileIn AS File

  'see if the file already exists, if so open it.
  IF Exist(filename) THEN
    hFileIn = OPEN filename FOR READ

    'read our single line of data
    'if there were more lines, we could set up a WHILE NOT Eof LOOP
    LINE INPUT #hFileIn, sInputLine

    'update the display data
    TextLabel1.Text = sInputLine

    'close the file
    CLOSE # hFileIn
  ELSE
```

A Beginner's Guide to Gambas – Revised Edition

```
' there was no .conf file to open
Message.Info("No preferences .conf file present.", "OK")
ENDIF
END
```

The reciprocal routine, WritePrefs() is shown below:

```
PUBLIC SUB WritePrefs( filename AS String)
'declare our local variables
DIM MyPath AS String
DIM hFileOut AS File
'note we are using the Dialog.Path value, not Application.Path
MyPath = Dialog.Path & filename
If Dialog.Path <> "" 'If a new path was set, create the file for write

    hFileOut = OPEN MyPath FOR WRITE CREATE
    ' print our data to the file
    PRINT # hFileOut, MyPath
    ' close the file
    CLOSE # hFileOut
ENDIF
END
```

As you may have already figured out, it is pretty simple but can easily be expanded to accommodate almost any set of configuration data your application may need to store. You are only limited by your imagination in Gambas. Now, let's look at our Format menu code. In the Format menu, the only options we have are to change colors or fonts. We are going to use the Qt dialogs to accomplish both tasks. Here is how it is done:

```
PUBLIC SUB FormatColorItem_Click()
'first, pop up a message to explain how we are going to work
Message.Info("Select Background color first, text color second.", "OK")

'set dialog caption to remind the user we are selecting BACKGROUND
Dialog.Title = " Select Background Color "

'call the standard dialog routine to select color
Dialog.SelectColor
'update our TextArea1.Backcolor with the value selected.
TextArea1.BackColor = Dialog.Color

'set dialog caption to remind the user we are selecting FOREGROUND
Dialog.Title = " Select Text Color "
```

```
'call the standard dialog routine to select color
Dialog.SelectColor

'update our TextArea1.ForeColor with the value selected.
TextArea1.ForeColor = Dialog.Color
END
```

Selecting the font and its attributes for our program is just as simple as it was to choose colors. Basically, we will set the dialog caption, call the standard Dialog.SelectFont routine, and update our TextArea1.Font property, like this:

```
PUBLIC SUB FormatFontItem_Click()
  Dialog.Title = " Select font "
  Dialog.SelectFont
  TextArea1.Font = Dialog.Font
END
```

This concludes our coding on Form1.Class. Now we return to Fmain.form. At the beginning of our program, in the constructor routine, we had to enable the timer. To actually make the timer do anything, you must program it. Double-clicking on the timer control will take you to the Timer1_Timer() subroutine, shown below. If you recall, we are going to have our program display the current time on one label and the current cursor position within the text area on another label. Here is how we accomplish that task:

```
PUBLIC SUB Timer1_Timer()
  'get the time for right now and convert to a string
  'to put in the TimeLabel.Text property
  TimeLabel.Text = Str$(Now)
  'we will get line and column data from the .Line and
  '.Column properties of the Label control and update the
  'PosLabel.Text property with their values:
  PosLabel.Text = "Line: " & Str(TextArea1.Line) & " Col: " & Str(TextArea1.Column)
END
```

The last menu is our Help menu. It has two options, Contents and About.

```
PUBLIC SUB HelpContentsItem_Click()
```

```
'for the "KDE" desktop, you should use this line
SHELL "konqueror file:/home/jon/Gambas_programs/FileOpsQT/help.html"
```

```
'for the Gnome desktop, the gnome-open command will open a folder or file, using the default browser.
SHELL "gnome-open /home/jon/Gambas_programs/FileOpsQT/help.html"
```

A Beginner's Guide to Gambas – Revised Edition

END

```
PUBLIC SUB HelpAboutItem_Click()
    Message.Info("<b>Simple Text Editor</b> by <br>John Rittinghouse, March, 2011.", "OK")
END
```



Chapter 10 – Math Operations

Gambas performs mathematical operations with a library of intrinsic and derived functions. According to the online encyclopedia Wikipedia¹⁴, “*an intrinsic function is a function available in a given language whose implementation is handled specially by the compiler. Typically, it substitutes a sequence of automatically-generated instructions for the original function call, similar to an inline function. Compilers that implement intrinsic functions generally enable them only when the user has requested optimization, falling back to a default implementation provided by the language runtime environment otherwise.*”

Intrinsic functions are built into the compiler of a programming language, usually to take advantage of specific CPU features that are inefficient to handle via external functions. The compiler's optimizer and code generator fully integrate intrinsic functions, which can result in some surprising performance speedups in calculation. Derived functions, on the other hand, are built in Gambas by using known mathematical formulas to apply specific algorithms that implement intrinsic functions. A table of derived functions and their corresponding algorithms can be found at the end of this chapter.

Precedence of Operations

When you use various operators in an expression in Gambas, there is a specific precedence of operations that Gambas uses to perform the operation. A well-defined protocol is followed:

- ✓ All expressions are simplified within parentheses from the inside outside
- ✓ All exponential operations are performed, proceeding from left to right
- ✓ All products and quotients are performed, proceeding from left to right
- ✓ All sums and differences are performed, proceeding from left to right

Bearing in mind the order of operations (i.e., the precedence), you can use the mathematical functions in Gambas to perform just about any mathematical operation needed for your programs. We will cover each function, either intrinsic or derived in this chapter and provide examples of how they are used. Table x at the end of this chapter provides you with the official “developer” hierarchy of operations, as defined in

Abs

Abs is an intrinsic function that returns the absolute value of a number. The absolute value of a number is its unsigned magnitude. For example, Abs(-1) and Abs(1) both return 1. The number argument can be any valid numeric expression. If number contains Null, an error is returned; if it is an initialized variable, zero is returned. Standard calling convention is:

¹⁴ See URL: http://en.wikipedia.org/wiki/Intrinsic_function.

Abs(number)

The following example uses the Abs function to compute the absolute value of a number. Try this on your console:

```
PUBLIC SUB MMain()
    DIM MyNum AS Variant
    MyNum = Abs(-1.03E3)
    PRINT MyNum
    MyNum = Abs( 2.5+-6.5)
    PRINT MyNum
END
```

Acs / ACos

Acs / ACos is a derived function that computes the inverse cosine (arc-cosine) of a number. $Y = \text{ACos}(X)$ returns the arc-cosine for each element of X . For real elements of X in the domain $[-1,1]$, $\text{ACos}(X)$ is real and in the range $[0, \pi]$. For real elements of X outside the domain $[-1,1]$, $\text{ACos}(X)$ is complex. The ACos function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are returned in radians. Syntax is:

```
value = ACos ( Number )
```

Example:

```
PRINT Acs(0.5)
1.047197551197
```

```
PRINT Acs(-1)
3.14159265359
```

Acsh / ACosh

The Acsh/ACosh is a derived function that computes the hyperbolic arc-cosine (inverse hyperbolic cosine) of a number X . It operates element-wise on arrays. All angles are in radians. The domain includes real numbers X such that $X \geq 1$. The range of the hyperbolic arc-cosine function is limited to non-negative real numbers Y such that $Y \geq 0$. Calling convention is as follows:

```
value = Acsh ( Number )
value = ACosh ( Number )
```

Here is an example to try on the console:

```
PUBLIC SUB Main()
  DIM y AS Variant
  Y = 2
  PRINT Acsh(Y)
END
```

The console responds with:

1.316957896925

Asn / ASin

Asn / ASin is a derived function that computes the arc-sine of a number. The ASin() function computes the principal value of the arc-sine of x. The value of x should be in the range [-1,1]. Upon successful completion, Asn() returns the arc sine of x, in the range [-π/2, π/2] radians. If the value of x is not in the range [-1,1] 0.0 is returned. The ASin() function will fail if the value x is not in the range [-1,1]. Standard calling convention is:

```
value = Asn ( Number )
value = ASin ( Number )
```

Here is an example:

```
PUBLIC SUB Main()
  DIM y AS Variant
  Y = 0.5
  PRINT Asn(Y)
  Y = -1.0
  PRINT Asn(Y)
END
```

The console responds with:

0.523598775598
-1.570796326795

Asnh / ASinh

Asnh / ASinh is a derived function that computes the inverse hyperbolic sine (*hyperbolic arc-sine*) for each element of X. The ASinh function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Standard calling convention is:

```
value = Asnh ( Number )
value = ASinh ( Number )
```

An example is as follows:

```
PUBLIC SUB Main()
DIM y AS Variant

Y = 2
PRINT Asnh(2)
END
```

The console responds with:

1.443635475179

Atn / ATan

Atn / ATan is an intrinsic function that computes the arc-tangent of a number. For real elements of X, ATan(X) is in the range $[-\pi/2, \pi/2]$. The ATan function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians. Standard calling convention is:

```
value = Atn ( Number )
value = ATan ( Number )
```

Example:

```
PUBLIC SUB Main()
DIM y AS Variant

Y = 0.5
PRINT ATan(Y)
END
```

The console responds with:

0.463647609001

Atnh / ATanh

Atnh/ATanh is a derived function that computes the inverse hyperbolic tangent (or *hyperbolic arc-tangent*) of a number. The function operates element-wise on arrays. It's domains and ranges include complex values. All angles are in radians. The calling convention of **Y = ATanh(X)** returns the hyperbolic arc-tangent for each element of X.

```
value = Atnh ( Number )
value = ATanh ( Number )
```

Here is an example you can try on the console using Atanh():

```
PUBLIC SUB Main()
    DIM MyResult AS Float
    DIM MyNum AS Float

    MyNum = 0.5
    MyResult = ATanh(0.5)
    PRINT "The hyperbolic arc-tangent of " & MyNum & " is: " & MyResult
END
```

The console responds with:

0.549306144334

Cos

Cos() is an intrinsic function that returns the cosine of an angle. The Cos function takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse. The result lies in the range -1 to 1. To convert degrees to radians, multiply degrees by $\pi/180$. To convert radians to degrees, multiply radians by $180/\pi$. Standard callin convention is:

```
Result = Cos(number)
```

The *number* argument can be any valid numeric expression that expresses an angle in radians. The following example uses the Cos function to return the cosine of an angle:

```
PUBLIC SUB Main()
    DIM MyAngle AS Float
    DIM MySecant AS Float

    MyAngle = 1.3          ' Define angle in radians.
```

A Beginner's Guide to Gambas – Revised Edition

```
MySecant = 1 / Cos(MyAngle) ' Calculate secant.  
PRINT "Secant of angle " & MyAngle & " (in radians) is: " & MySecant  
END
```

The console responds with:

```
Secant of angle 1.3 (in radians) is: 3.738334127075
```

Cosh

Cosh is a derived function that computes the hyperbolic cosine of a number. The Cosh function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians. Standard calling convention is:

Value = Cosh (Number)

Here is an example to try on the console:

```
PUBLIC SUB Main()  
DIM MyAngle AS Float  
DIM MyResult AS Float  
  
MyAngle = 1.0 ' Angle expressed in radians.  
MyResult = Cosh(MyAngle)  
PRINT "Cosh() of angle " & MyAngle & " (in radians) is: " & MyResult  
END
```

The console responds with:

```
Cosh() of angle 1 (in radians) is: 1.543080634815
```

Deg and Rad

Deg is an intrinsic conversion function that converts radians to degrees. Rad is a reciprocal conversion function that converts degrees to radians. The calling convention for Deg and Rad is:

Value = Deg (Angle)
Value = Rad (Angle)

An example to try on the console is:

```
PUBLIC SUB Main()
```

```
PRINT Deg(Pi / 2)
PRINT Rad(90)
PRINT Rad(180) - Pi
END
```

The console responds with:

```
90
1.570796326795
□
```

Exp

Exp is an intrinsic function that computes the exponential of a number. It returns e (the base of natural logarithms, where the constant e is approximately 2.718282) raised to a power. The number argument can be any valid numeric expression. If the value of number exceeds 709.782712893, an error occurs. The Exp function complements the action of the Log function and is sometimes referred to as the anti-logarithm. The calling convention is:

Value = Exp (Number)

Here is an example:

```
PUBLIC SUB Main()
PRINT Exp(1)
END
```

The console responds with:

```
2.718281828459
```

Here is another example that uses the Exp function to return e raised to a power:

```
PUBLIC SUB Main()
DIM MyAngle AS Float
DIM MyHSin AS Float ' Define angle in radians.
MyAngle = 1.3 'Calculate the hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
PRINT MyHSin
END
```

The console responds with:

1.698382437293

Fix and Frac

Fix is an intrinsic function that returns the integer part of a number. Frac is an intrinsic function that computes the fractional part of a number. Note that in the example below, Frac() returns the absolute value of the fractional result of $-\pi$. The standard calling convention is:

Value = Fix (Number)
Value = Frac (Number)

Example:

```
PUBLIC SUB Main()
PRINT Fix(Pi)
PRINT Frac(Pi)

PRINT Fix(-Pi)
PRINT Frac(-Pi)
END
```

The console responds with:

```
3
0.14159265359
-3
0.14159265359
```

Int

Int(*Number*) returns the mathematical integer part of a number, i.e., the greater integer smaller than *Number*.

Value = Int (Number)

Here are some examples you can try on the console:

```
PRINT Int(Pi)
3

PRINT Int(-Pi)
-4
```

Log

Log is an intrinsic function that computes the natural logarithm of e . The logarithm having base e . The constant e is approximately 2.718282 where e is a unique number with the property that the area of the region bounded by the hyperbola $y = 1/x$, the x-axis, and the vertical lines $x=1$ and $x=e$ is 1. The notation $\ln x$ is used in physics and engineering to denote the natural logarithm, while mathematicians commonly use the notation $\log x$. The *Number* argument can be any valid numeric expression greater than 0. Standard calling convention is:

value = Log (Number)

Here is an example:

```
PUBLIC SUB Main()
PRINT Log(2.71828)
PRINT Log(1)
END
```

The console responds with:

0.9999999327347

□

You can create a derived function to calculate base-n logarithms for any number x by dividing the natural logarithm of x by the natural logarithm of n as follows:

$\text{Logn}(x) = \text{Log}(x) / \text{Log}(n)$

Here is an example to try:

```
PUBLIC SUB Main()
DIM n AS Float
DIM x AS Float
DIM logn AS Float

n = 2
x = 10
logn = Log(x) / Log(n)
PRINT logn & " is the result of Logn(" & x & ")"
END
```

The console responds with:

3.321928094887 is the result of Log(10)

Log10

Log10 is a derived function that computes the decimal logarithm of a number. Log10(x) = Log(x) / Log(10). Calling convention is:

value = Log10 (Number)

Example :

```
PRINT Log10(10)  
1
```

Max and Min

Max returns the greater expression of the list. Expressions must be numbers or date/time values. Min returns the smaller expression of the list. Expressions must be numbers or date/time values.

value = Max (Expression [, Expression ...])
value = Min (Expression [, Expression ...])

Examples

```
PUBLIC SUB Main()  
PRINT Max(6, 4, 7, 1, 3)  
PRINT Max(Now, CDate("01/01/1900"), CDate("01/01/2100"))  
PRINT Min(6, 4, 7, 1, 3)  
PRINT  
PRINT Min(Now, CDate("01/01/1900"), CDate("01/01/2100"))  
END
```

The console responds with:

```
7  
01/01/2100  
  
1  
01/01/1900
```

Pi

Pi returns $\pi * Number$. If *Number* is not specified, it is assumed to be one. Standard calling convention is:

Result = Pi ([Number])

Examples:

```
PRINT Pi  
3.14159265359
```

```
PRINT Pi(0.5)  
1.570796326795
```

Randomize and Rnd

Randomize Initializes the pseudo-random number generator with a methodology that uses the current date and time. The calling convention is:

Randomize (Number)

where the number argument can be any valid numeric expression. Randomize uses *Number* to initialize the Rnd function's random-number generator, giving it a new seed value. If you omit number, the value returned by the system timer is used as the new seed value. If Randomize is not used, the Rnd function (with no arguments) uses the same number as a seed the first time it is called, and thereafter uses the last generated number as a seed value. In other words, for any given initial seed, the same number sequence is generated because each successive call to the Rnd function uses the previous number as a seed for the next number in the sequence.

In order to repeat sequences of random numbers, call Rnd with a negative argument immediately before using Randomize with a numeric argument. Before calling Rnd, use the Randomize statement without an argument to initialize the random-number generator with a seed based on the system timer. Using Randomize with the same value for number does not repeat the previous sequence.

Rnd computes a pseudo-random floating point number, using the Lehmer algorithm. If no parameters are specified, Rnd returns a pseudo-random number in the interval [0 , 1]. If only one parameter is specified, Rnd returns a pseudo-random number in the interval [0, Min]. If both parameters are specified, Rnd returns a pseudo-random number in the interval [Min, Max]. Standard calling convention for Rnd is:

Rnd ([Min [, Max])

The number argument can be any valid numeric expression. The Rnd function returns a value less than 1 but greater than or equal to 0. The value of number determines how Rnd generates a random number. Here is a table that helps explain it:

Random Function Determination Table

If number is	Rnd generates
Less than zero	The same number every time, using number as the seed.
Greater than zero	The next random number in the sequence.
Equal to zero	The most recently generated number.
Not supplied	The next random number in the sequence.

To produce random integers that fall within any given range, use this formula:

Int((upperbound - lowerbound + 1) * Rnd + lowerbound)

Here, upperbound is the highest number in the range, and lowerbound is the lowest number in the range. Here is an example program to try on the console:

```
PUBLIC SUB Main()
DIM Dice AS Integer

PRINT "Between 0 and 1: ";
PRINT Rnd
PRINT "Between 0 and 2: ";
PRINT Rnd(2)
PRINT "Between Pi and Pi*2: ";
PRINT Rnd(Pi, Pi(2))
Randomize
DO WHILE Dice <> 1
    Dice = Int(Rnd(1,7))
    'Throws the dice between 1 and 6
    PRINT "You threw a " & dice
LOOP
END
```

The console responds with:

```
Between 0 and 1: 7.826369255781E-6
Between 0 and 2: 0.263075576164
Between Pi and Pi*2: 5.515396781706
You threw a 6
```

```
You threw a 4  
You threw a 1
```

Round

Round rounds a number to its nearest integer if Digits is not specified. If Digits is specified, rounds to 10^{\wedge} Digits .

Value = Round (Number [, Digits])

Example:

```
PUBLIC SUB Main()  
PRINT Round(Pi, -2)  
PRINT Round(1972, 2)  
END
```

The console responds with:

```
3.14  
2000
```

Sgn

Sgn returns an integer indicating the sign of a number. If the number is zero, it returns zero. If the number is strictly positive, it returns the integer number +1. If the number is strictly negative, it returns the integer number -1. Calling convention is:

Sign = Sgn (Number)

Example:

```
PUBLIC SUB Main()  
PRINT Sgn(Pi)  
PRINT Sgn(-Pi)  
PRINT Sgn(0)  
END
```

The console responds with:

```
|
```

-|
□

Sin

Sin is an intrinsic function that computes the sine of an angle. The angle is specified in radians. The sine function $\sin x$ is one of the basic functions encountered in trigonometry (the others being the cosecant, cosine, cotangent, secant, and tangent). Let θ be an angle measured counterclockwise from the x-axis along an arc of the unit circle. Then $\sin \theta$ is the vertical coordinate of the arc endpoint.

As a result of this definition, the sine function is periodic with period 2π . By the Pythagorean theorem, $\sin \theta$ also obeys the identity $\sin^2\theta + \cos^2\theta = 1$. The standard calling convention for Sin is:

Value = Sin (Angle)

Example:

```
PUBLIC SUB Main()
    PRINT Sin(Pi/2)
END
```

The console responds with:

|

Sinh

Sinh is a derived function that computes the hyperbolic sine of a number. Standard calling convention for Sinh is:

Value = Sinh (Number)

The following example uses the Exp function to calculate hyperbolic sine of an angle:

```
PUBLIC SUB Main()
    DIM MyAngle AS Float
    DIM MyHSin AS Float ' Define angle in radians
    MyAngle = 1.3 ' Calculate hyperbolic sine.
    MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
    PRINT MyHSin
END
```

The console responds with:

1.698382437293

Sqr

Sqr is an intrinsic function that returns the square root of a number. The number argument can be any valid numeric expression greater than or equal to 0 (non-negative). The calling convention is:

Value = Sqr (Number)

Here is an example:

PRINT Sqr(2)

1.414213562373

The following example uses the Sqr function to calculate the square root of a number:

```
PUBLIC SUB Main()
  DIM MySqr AS Float
  MySqr = Sqr(4)  ' Returns 2.
  PRINT MySqr & " is the result of Sqr(4)"
  MySqr = Sqr(23) ' Returns 4.79583152331272.
  PRINT MySqr & " is the result of Sqr(23)"
  MySqr = Sqr(0)  ' Returns 0.
  PRINT MySqr & " is the result of Sqr(0)"
  'MySqr = Sqr(-4) ' Generates a run-time error.
  'PRINT MySqr & " is the result of Sqr(-4)"
END
```

Tan

Tan is an intrinsic function that computes the tangent of an angle. The angle argument can be any valid numeric expression that expresses an angle in radians. Tan takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle. To convert degrees to radians, multiply degrees by pi /180. To convert radians to degrees, multiply radians by 180/pi. Standard calling convention is:

Value = Tan (angle)

The following example uses the Tan function to return the tangent of an angle:

```
PUBLIC SUB Main()
DIM MyAngle AS Float
DIM MyCotangent AS Float
MyAngle = 1.3 ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle) ' Calculate cotangent.
PRINT MyCotangent
PRINT Tan(Pi/4)
END
```

The console responds with:

```
0.277615646541
```

Tanh

Tanh is a derived function that computes the hyperbolic tangent of a number. The standard calling convention for Tanh is:

Value = Tanh (Number)

Example:

```
PUBLIC SUB Main()
PRINT Tanh(1)
END
```

The console responds with:

```
0.761594155956
```

The following non-intrinsic math functions are provided in Gambas as built-in intrinsic math functions:

Gambas Derived Math Functions

Function	Derived equivalents
Inverse Sine	$\text{ASin}(X) = \text{Atn}(X / \text{Sqr}(-X * X + 1))$
Inverse Cosine	$\text{ACos}(X) = \text{Atn}(-X / \text{Sqr}(-X * X + 1)) + 2 * \text{Atn}(1)$
Hyperbolic Sine	$\text{Sinh}(X) = (\text{Exp}(X) - \text{Exp}(-X)) / 2$
Hyperbolic Cosine	$\text{Cosh}(X) = (\text{Exp}(X) + \text{Exp}(-X)) / 2$
Hyperbolic Tangent	$\text{Tanh}(X) = (\text{Exp}(X) - \text{Exp}(-X)) / (\text{Exp}(X) + \text{Exp}(-X))$

A Beginner's Guide to Gambas – Revised Edition

Inverse Hyperbolic Sine	$\text{ASinh}(X) = \text{Log}(X + \text{Sqr}(X * X + 1))$
Inverse Hyperbolic Cosine	$\text{ACosh}(X) = \text{Log}(X + \text{Sqr}(X * X - 1))$
Inverse Hyperbolic Tangent	$\text{ATanh}(X) = \text{Log}((1 + X) / (1 - X)) / 2$
Logarithm to base 10	$\text{Log10}(X) = \text{Log}(X) / \text{Log}(N)$

Other Derived Math Functions

Secant	$\text{Sec}(X) = 1 / \text{Cos}(X)$
Cosecant	$\text{Cosec}(X) = 1 / \text{Sin}(X)$
Cotangent	$\text{Cotan}(X) = 1 / \text{Tan}(X)$
Inverse Secant	$\text{Arcsec}(X) = \text{Atn}(X / \text{Sqr}(X * X - 1)) + \text{Sgn}(X - 1) * (2 * \text{Atn}(1))$
Inverse Cosecant	$\text{Arccosec}(X) = \text{Atn}(X / \text{Sqr}(X * X - 1)) + (\text{Sgn}(X) - 1) * (2 * \text{Atn}(1))$
Inverse Cotangent	$\text{Arccotan}(X) = \text{Atn}(X) + 2 * \text{Atn}(1)$
Hyperbolic Secant	$\text{HSec}(X) = 2 / (\text{Exp}(X) + \text{Exp}(-X))$
Hyperbolic Cosecant	$\text{HCosec}(X) = 2 / (\text{Exp}(X) - \text{Exp}(-X))$
Hyperbolic Cotangent	$\text{HCotan}(X) = (\text{Exp}(X) + \text{Exp}(-X)) / (\text{Exp}(X) - \text{Exp}(-X))$
Inverse Hyperbolic Secant	$\text{HArcsec}(X) = \text{Log}((\text{Sqr}(-X * X + 1) + 1) / X)$
Inverse Hyperbolic Cosecant	$\text{HArcosec}(X) = \text{Log}((\text{Sgn}(X) * \text{Sqr}(X * X + 1) + 1) / X)$
Inverse Hyperbolic Cotangent	$\text{HArccotan}(X) = \text{Log}((X + 1) / (X - 1)) / 2$
Logarithm to base N	$\text{LogN}(X) = \text{Log}(X) / \text{Log}(N)$

Since Gambas does not directly provide functions for these specific derived functions, we are going to build a module that will extend Gambas capabilities by providing all of the functions listed in the table above as a final exercise in this chapter. Our module will be named Derived and you will be able to access all of the functions in your code by using the format `Derived.function_name` where `function_name` will be one of the functions we build in the next section.

Start Gambas and create a terminal application. MMain is your default startup class. Now create a new module and name it Derived. When the new code window opens for the Derived module, go back to MMain and enter the following:

```
' Gambas class file
```

```
PUBLIC SUB Main()
DIM myVal AS Float
```

```
myVal = 51.5
WHILE myVal <> 0.0
```

```
PRINT "Enter a number > 1 or enter 0 to quit: ";
LINE INPUT myVal
```

```
IF myVal = 0.0 THEN BREAK  
  
IF myVal <= 1.0 THEN  
PRINT"Number was <= 1.0"  
CONTINUE  
ENDIF  
  
PRINT Derived.Sec(myVal) & " is secant"  
PRINT Derived.CoSec(myVal) & " is cosecant"  
PRINT Derived.CoTan(myVal) & " is cotangent"  
PRINT  
PRINT Derived.ArcSec(myVal) & " is arcsecant"  
PRINT Derived.Arccosec(myVal) & " is arccosecant"  
PRINT Derived.Arccotan(myVal) & " is arccotangent"  
PRINT  
PRINT Derived.HSec(myVal) & " is hyperbolic secant"  
PRINT Derived.HCoSec(myVal) & " is hyperbolic cosecant"  
PRINT Derived.HCoTan(myVal) & " is hyperbolic cotangent"  
PRINT  
PRINT Derived.HArcSec(myVal) & " is hyperbolic arcsecant"  
PRINT Derived.HArcCoSec(myVal) & " is hyperbolic arccosecant"  
PRINT Derived.HArcCoTan(myVal) & " is hyperbolic arccotangent"  
PRINT  
PRINT Derived.LogN(myVal, 2) & " is LogN of: " & myVal & " to base 2"  
PRINT "of myVal: " & myVal  
WEND  
END
```

Now, here is the code for the Derived.module file:

```
' Gambas Derived.module file  
PUBLIC FUNCTION Sec(x AS Float) AS Float  
DIM result AS Float  
result = 1.0 / Cos(x)  
RETURN result  
END  
  
PUBLIC FUNCTION CoSec(x AS Float) AS Float  
DIM result AS Float  
result = 1.0 / Sin(x)  
RETURN result  
END
```

```
PUBLIC FUNCTION CoTan(x AS Float) AS Float
```

```
    DIM result AS Float
```

```
    result = 1.0 / Tan(x)
```

```
    RETURN result
```

```
END
```

```
PUBLIC FUNCTION ArcSec(x AS Float) AS Float
```

```
    DIM result AS Float
```

```
    result = Atn(x/Sqr(x*x-1))+(Sgn(x)-1)*(2*Atn(1))
```

```
    RETURN result
```

```
END
```

```
PUBLIC FUNCTION Arccosec(x AS Float) AS Float
```

```
    DIM result AS Float
```

```
    result = Atn(x/Sqr(x*x-1))+(Sgn(x)-1)*(2*Atn(1))
```

```
    RETURN result
```

```
END
```

```
PUBLIC FUNCTION Arccotan(x AS Float) AS Float
```

```
    DIM result AS Float
```

```
    result = Atn(x)+(2*Atn(1))
```

```
    RETURN result
```

```
END
```

```
PUBLIC FUNCTION HSec(x AS Float) AS Float
```

```
    DIM result AS Float
```

```
    result = 2.0 / (Exp(x) + Exp(-x))
```

```
    RETURN result
```

```
END
```

```
PUBLIC FUNCTION HCoSec(x AS Float) AS Float
```

```
    DIM result AS Float
```

```
    result = 2.0 / (Exp(x) - Exp(-x))
```

```
    RETURN result
```

```
END
```

```
PUBLIC FUNCTION HCoTan(x AS Float) AS Float
```

```
    DIM result AS Float
```

```
    result = (Exp(x) + Exp(-x))/(Exp(x) - Exp(-x))
```

```
    RETURN result
```

```
END
```

A Beginner's Guide to Gambas – Revised Edition

```
'this routine currently produces an error in Gambas  
'if you try to use the formula result = Log((Sqr(-x*(x+l))+l)/x)  
'so this is a work-around to get the same results
```

```
PUBLIC FUNCTION HArcsec(x AS Float) AS Float
```

```
    DIM result AS Float
```

```
    DIM Temp1 AS Float
```

```
    DIM Temp2 AS Float
```

```
    DIM Temp3 AS Float
```

```
    Temp1 = ((x * -l)*( x + l))+l
```

```
    Temp2 = Sqr(Abs(Temp1))
```

```
    Temp3 = Log(Temp2)
```

```
    result = Temp3 * -l
```

```
'this call will generate an error because the
```

```
'Sqr function will not work with negative values
```

```
'result = Log((Sqr(-x*(x+l))+l)/x)
```

```
    RETURN result
```

```
END
```

```
PUBLIC FUNCTION HArcosec(x AS Float) AS Float
```

```
    DIM result AS Float
```

```
    result = Log((Sgn(x)*Sqr(x*x+l)+l)/x)
```

```
    RETURN result
```

```
END
```

```
PUBLIC FUNCTION HArcctan(x AS Float) AS Float
```

```
    DIM result AS Float
```

```
    result = Log((x+l)/(x-l))/2
```

```
    RETURN result
```

```
END
```

```
PUBLIC FUNCTION LogN(x AS Float, n AS Float) AS Float
```

```
    DIM result AS Float
```

```
    result = Log(x)/Log(n)
```

```
    RETURN result
```

```
END
```

When you run the program, you will a prompt to enter a number > 1 or 0 to quit. Enter 4 and you should see the following results:

A Beginner's Guide to Gambas – Revised Edition

Enter a number > 1 or enter 0 to quit: 4

-1.529885656466 is secant

-1.321348708811 is cosecant

0.863691154451 is cotangent

0.85707194785 is arcsecant

0.801529994232 is arccosecant

2.896613990463 is arccotangent

0.036618993474 is hyperbolic secant

0.036643570326 is hyperbolic cosecant

1.000671150402 is hyperbolic cotangent

-1.472219489583 is hyperbolic arcsecant

0.247466461547 is hyperbolic arccosecant

0.255412811883 is hyperbolic arccotangent

2 is LogN of: 4 to base 2

of myVal: 4

Enter a number > 1 or enter 0 to quit:

A Beginner's Guide to Gambas – Revised Edition

The **gb_reserved_temp.h** source file where operator precedence is defined in Gambas.

```
// Operator  Type   Value  Hierarchy ByteCode
{{"=",    RSF_OP2S,  OP_EQUAL,  4,  C_EQ  },
 {"==",   RSF_OP2S,  OP_NEAR,   4,  C_NEAR },
 {"(",    RSF_OPP,   OP_LBRA,   12,   },
 {"()",   RSF_OPP,   OP_RBRA,   12,   },
 {".",    RSF_OP2,   OP_PT,     20,   },
 {"!",    RSF_OP2,   OP_EXCL,   20,   },
 {"+",    RSF_OP2,   OP_PLUS,   5,  C_ADD },
 {"-",    RSF_OP2,   OP_MINUS,  5,  C_SUB },
 {"*",    RSF_OP2,   OP_STAR,   6,  C_MUL },
 {"/",    RSF_OP2,   OP_SLASH,  6,  C_DIV },
 {"^",    RSF_OP2S,  OP_FLEX,   7,  C_POW },
 {"&",   RSF_OPN,   OP_AMP,    8,  C_CAT },
 {">",   RSF_OP2S,  OP_GT,     4,  C_GT  },
 {"<",   RSF_OP2S,  OP_LT,     4,  C_LT  },
 {">=",  RSF_OP2S,  OP_GE,     4,  C_GE  },
 {"<=",  RSF_OP2S,  OP_LE,     4,  C_LE  },
 {"<>",  RSF_OP2S,  OP_NE,     4,  C_NE  },
 {"|",    RSF_OPP,   OP_LSQR,   12,   },
 {"||",   RSF_NONE,  OP_RSQR,   12,   },
 {"AND",  RSF_OP2SM, OP_AND,    2,  C_AND },
 {"OR",   RSF_OP2SM, OP_OR,     2,  C_OR  },
 {"NOT",  RSF_OPI,   OP_NOT,    10, C_NOT },
 {"XOR",  RSF_OP2SM, OP_XOR,    2,  C_XOR },
 {"\\\",  RSF_OP2S,  OP_DIV,    6,  C_QUO },
 {"DIV",  RSF_OP2S,  OP_DIV,    6,  C_QUO },
 {"MOD",  RSF_OP2S,  OP_MOD,    6,  C_Rem },
 {"LIKE", RSF_OP2S,  OP_LIKE,   4,  C_LIKE },
 {"&/",  RSF_OPN,   OP_FILE,   9,  C_FILE }}
```



Chapter 11 – Object-Oriented Concepts

In the field of computer science, object-oriented programming (OOP) is considered a computer programming paradigm. The general idea is that a computer program is composed of a collection of individual units (called objects), each of which is capable of receiving messages, processing data, and sending messages to other units or objects. This differs from the traditional view of programming where a program is seen as a list of instructions that are executed in a sequential order by the computer.

Proponents of object-oriented programming have claimed the approach gives programmers more flexibility and facilitates making changes to programs when needed. The paradigm is very popular in companies and government entities that conduct software engineering and development as a matter of course. Furthermore, many people believe OOP to be easier to learn for novice programmers when compared with other languages such as Java, C++, etc. OOP, they claim, is simpler to develop and maintain and that it lends itself to a better understanding of complex situations and procedures than other programming methods.

There are almost as many critics of the OO paradigm as there are proponents. Some of the most common arguments critics raise are that OO-based languages like Smalltalk, Lisp, and others have had their progress stalled with the advent of newer languages like C++, C#, and Java. Small, interpreted languages like Tcl, Perl, and Python have developed a fairly large following but they are not making any progress in the development of a true OO paradigm. Critics also have said the object-oriented approach does not adequately address future computing requirements. They feel object-oriented languages have sacrificed simplicity—the very thing that makes programming approachable to a wider audience. Java, for example, is an extremely complex implementation of OO concepts. Others contend concepts like encapsulation and inheritance, which were originally intended to “save” programmers from themselves while developing software, have failed because they implement global properties. Modularity is another way of keeping things local in scope so people can understand code better. That was what encapsulation was supposed to accomplish. Today, open source software seems to have evolved a workable approach to the problem of code maintenance, although it is not without its own issues.

Objects promised reuse, and the programming community has not seen much success with that. The ability to maintain a program once it has been written and to do localized debugging and to do much larger development efforts are also reasons cited as significant advantages gained with the use of OOPLs. In the 1990s, over-optimism regarding the benefits of OO programming led businesses to expect miracles. When software developers could not deliver on the outrageous business plans based on those promises, 2001 saw businesses fall like dominos, leaving the entire tech industry in a recession. Defunct companies littered Silicon Valley like like so many bleached bones in a desert.

Fortunately, Gambas takes a very pragmatic approach to the implementation of OO

concepts, leveraging what works and, unlike C++, avoiding that which will not ever work well. The GB implementation is an elegant compromise between true OO theory and the business realities programmers have to contend with today.

OOP is often called a paradigm rather than a style or type of programming to emphasize the point that OOP can change the way software is developed by changing the way that programmers and software engineers think about software. Much of the evolution of any language depends on the OS that supports that language. With the advent of Linux and the huge ground swell of support for open source software, the future looks very bright for Gambas. Now, let's take a look at some of the basic OO concepts that GB supports.

Fundamentals of Object Oriented Programming

Object-oriented programming is based upon several key concepts: objects, data abstraction, encapsulation, polymorphism, and inheritance. One distinguishing feature of OOP is the methods used for handling data types. *Object-type* data is generally required to satisfy programmer-defined constraints. A data-type restricted to a specific object is considered to be a **private** data-type. This type of language-imposed constraint restricting data types to specific objects forces programmers (by design) to code without relying on creating a laundry list of variables visible to every object in the program (global variables). Variables can be used and disposed of within the object or class in which they are needed and they do not affect other parts of a program. The actions taken by the program where such variables are used are referred to as methods. In more traditional languages, methods equate to functions or subroutines. In OOP, these constraints also may extend to the use of methods and are not exclusive to variables.

Object-oriented languages provide internal mechanisms for ensuring that variables and methods are only visible and available within the scope of the class or method in which they are declared. Albeit, some variables and methods are designed to be visible to all other methods or classes of a project. In those instances, they are referred to as **public** variables and methods. In Gambas, variables and methods that are public are global in scope *to the class* in which they are declared. Variables meant to be visible only to the methods in which they are declared as **private** within a given class are created with the DIM keyword in Gambas.

Objects

Objects are comprised of all the data and functionality that exist within distinct units in a running (executing) computer program. Objects are the foundation of modularity and structure in an OOP. They have two basic attributes: they are self-contained and they are uniquely identifiable. Having these two attributes allows all of the various program components to correlate variables in a program with various real-world aspects of a problem for which a program is intended to solve.

Data Abstraction

Data Abstraction - Data abstraction is a methodology used in OOPs that enables a programmer to specify how a data object is used. Abstraction allows a programmer to isolate data objects from the underlying details of how the data is created (usually by using even more primitive data objects). In other words, the use of a data object, comprised of one or more primitive data objects, allows a programmer to structure his or her code to use data objects that operate on *abstract* data.

Data objects that are comprised of one or more primitive data objects are called compound data objects. With abstraction, the program can use data without making any assumptions about it (i.e., the underlying details of how or where that data came from) that isn't necessary for performing a given task. A *concrete* data representation is defined independent of any programs that use the data. The interface between the concrete and abstract data is a set of procedures, called constructors, that implement abstract data in terms of the concrete representation. Methods (subroutines and functions) may also be so abstracted in the OOP paradigm.

Encapsulation

Encapsulation - Ensures that users of an object cannot change the internal state of the object in unexpected ways; only the object's internally defined methods are allowed to access its state (or attributes). Each object exposes an interface that specifies how other objects may interact with it. Other objects will not know of this object's internal methods and must therefore rely upon this object's interface in order to interact with it.

Polymorphism

Polymorphism means the ability to take more than one form. In OOPs, it refers to a programming language's ability to process objects differently depending on the data type or class. More specifically, polymorphism is the ability to redefine methods for derived classes. An operation may exhibit different behaviors in different circumstances. The behavior is dependent upon the data type used in the operation. For example, given a base class named shape, polymorphism enables the programmer to define different methods of area calculation for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the area method to it will return the correct results. Polymorphism is used extensively in OOPs when implementing *inheritance* and is considered to be a requirement of any true OOPL.

Inheritance

Inheritance is the process by which objects can acquire the properties of objects that exist in another class. In Gambas, all of the controls of the Toolbox inherit the properties of the class named Control. In OOP, inheritance provides reusability and extensibility (adding additional features to an existing class without the need to modify it). This is achieved by

deriving a new class from an existing class and making changes to the new *instance* of that class. The new class will have combined the features of both classes. Inheritance allows objects to be defined and created that are specialized implementations of already-existing objects. They can share (and extend) the existing object's behavior without having to reimplement it.

The Gambas Approach to OOP

An OO program is generally designed by identifying all of the objects that will exist in a system. The code which actually does the work is irrelevant to the object due to encapsulation. The biggest challenge in OOP is of designing an object system that is understandable and maintainable. It should be noted that there are distinct parallels between the object-oriented paradigm and traditional systems theory. While OOP focuses on objects as units in a system, systems theory focuses on the system itself. Somewhere in between, one may find software design patterns or other techniques that use classes and objects as building blocks for larger components. Such components can be seen as an intermediate step from the object-oriented paradigm towards the more pragmatically oriented models of systems theory. Such appears to be the Gambas approach to OOP. It is an elegant, pragmatic solution that takes the best of both approaches and makes programming easy and simple.

Gambas has tried to keep simplicity at the forefront of an approach that leverages OO concepts and compromises with systems theory where it makes good sense. In that regard, Gambas has taken a *class-based approach* to OOP. The class-based OOP approach is an OOP paradigm where the concept of a class is central. In this approach encapsulation prevents users from breaking invariants of the class. Class invariants are established during construction and are constantly maintained between calls to public methods. Temporarily breaking class invariance between private method calls is possible, although it is not encouraged. Sometimes, it is useful because it allows the implementation of a class of objects to be changed for aspects not visible to or exposed inside the class interface without having an impact on user code that may have implemented features of that class.

The definition of encapsulation in a class-based OOP paradigm focuses on grouping and packaging of related information (*cohesion*) rather than on program security issues. OOP languages do not normally offer formal security restrictions to the internal object state. Using a method of access is a matter of convention for the interface design. Inheritance is typically done by grouping objects into classes and defining classes as extensions of existing classes. This grouping of classes is sometimes organized into tree or lattice structures, reflecting behavioral commonality.

Gambas Classes

In Gambas, you can define your own classes. Remember that a class is merely template that is used by the program at runtime. The class itself is not used directly. To use a class, you must make a variable declaration of the class data-type, create a copy of the class for the

program to use, then instantiate the copy. The instantiated copy is called an object. Since a class is a real data-type, the declaration of the object looks like any other normal variable declaration:

object_name AS Class_name

Instantiating an object means to create a new instance of the template and assign that instance the name on the left side of the equal sign. In GB, this is accomplished using the **NEW** keyword:

object_name = NEW Class_name

When *object_name* is instantiated as a new copy of the template defined by *Class_name*, the programming language allows *object_name* to inherit all of the methods and attributes of *Class_name*. When the new object is created, it automatically invokes a **constructor** to initialize itself with default variables.

Sample program: Contacts

To illustrate the use of a class in Gambas, we are going to create a sample program that implements a contacts book. It is going to be fairly simple but will be useful to maintain contacts. To begin, create a new project named Classes and when you get to the IDE, create a class (not a startup class) named Contact. This is where we will start with our program, building a class definition for a Contact.

The Contact class

```
' Gambas Contact.class file
PUBLIC FirstName AS String
PUBLIC LastName AS String
PUBLIC Initial AS String
PUBLIC Suffix AS String
PUBLIC Street1 AS String
PUBLIC Street2 AS String
PUBLIC City AS String
PUBLIC State AS String
PUBLIC Zip5 AS String
PUBLIC Zip4 AS String
PUBLIC Areacode AS String
PUBLIC Prefix AS String
PUBLIC Last4 AS String
PUBLIC RecordID AS Integer
```

We want our class to provide two methods, one to retrieve our contact data from the file it is stored in and one to put data into that file. Appropriately, the two methods will be named *Contact.GetData* and *Contact.PutData*.

Contact.GetData Method

Let's start with the GetData method, which is decleared as a function because it will take a collection variable as a parameter and return a collection **filled** with data from the *contacts.data* file if that data exists. We will declare local variables within our methods by using the DIM keyword. Note that we could declare these variables as PRIVATE outside of the method definitions and it would render them visible only to the methods within the contact class. However, declarations local to the method themselves further restricts the scope and makes the entire class file cleaner in appearance from a user perspective. They will see all the public methods and variables and nothing more.

The collection variable is passed in as a parameter from the constructor call in FMain.class so it is not necessary to declare it separately. The function will return a collection variable so we add **AS Collection** to the end of the function declaration. We will show how the collection variable *cTheData* is passed to this function when it is called from the FMain.class file.

PUBLIC FUNCTION *GetData (cTheData AS Collection) AS Collection*

'this string var will hold each line of data read from the file

DIM *sInputLine AS String*

'we need a file handle and two string vars for building filenames

DIM *hFileIn AS File*

DIM *filename AS String*

DIM *fullpath AS String*

'this class variable is used to convert the string data we read in

'from file into a contact record. We declare the variable here and

'will instantiate it below

DIM *MyContactRecord AS Contact*

'a local integer var to keep track of how many records we added

DIM *RecordNum AS Integer*

'this variable will take the string read in and convert all the fields

'that are comma-delimited into an array using the split statement

DIM *arsInputFields AS String[]*

'this counter is used to loop through the array and assign values

'to the contact records

A Beginner's Guide to Gambas – Revised Edition

```
DIM counter AS Integer
```

```
filename = "contacts.data" 'hardcoded because it will not change
```

```
'we will use the current dir where the app is to build full pathname  
fullpath = Application.Path & filename
```

```
'now, see if the file already exists, if so open it for READ.
```

```
IF Exist(fullpath) THEN  
OPEN fullpath FOR READ AS #hFileIn
```

```
'start with record 1
```

```
RecordNum = 1
```

```
'we will loop through every line of data until we reach End of file
```

```
WHILE NOT Eof(hFileIn)
```

```
    'every contact added to the collection must be instantiated  
    'each time a record is added to the collection. You CAN  
    'reuse the same class var for each re-instantiation though.  
    MyContactRecord = NEW Contact 'here is where it renews itself
```

```
'read the line in from file and store in string var sInputLine
```

```
LINE INPUT #hFileIn, sInputLine
```

```
'used for console debugging
```

```
PRINT "Reading: " & sInputLine
```

```
'now, fill our array of strings using Split to convert the line
```

```
'of string data read in to fields in an array
```

```
arsInputFields = Split(sInputLine, ",")
```

```
'for debug only to show that all fields convert correctly
```

```
FOR counter = 0 TO 14
```

```
    PRINT "Loading: " & arsInputFields[counter]
```

```
NEXT
```

```
'now, we put the field data into our local contact class var
```

```
'and will add that contact record to the collection of contacts
```

```
MyContactRecord.FirstName = arsInputFields[0]
```

```
MyContactRecord.Initial = arsInputFields[1]
```

```
MyContactRecord.LastName = arsInputFields[2]
```

```
MyContactRecord.Suffix = arsInputFields[3]
```

```
MyContactRecord.Street1 = arsInputFields[4]
```

A Beginner's Guide to Gambas – Revised Edition

```
MyContactRecord.Street2 = arsInputFields[5]
MyContactRecord.City = arsInputFields[6]
MyContactRecord.State = arsInputFields[7]
MyContactRecord.Zip5 = arsInputFields[8]
MyContactRecord.Zip4 = arsInputFields[9]
MyContactRecord.Areacode = arsInputFields[10]
MyContactRecord.Prefix = arsInputFields[11]
MyContactRecord.Last4 = arsInputFields[12]
MyContactRecord.RecordID = CInt(arsInputFields[13])
MyContactRecord.Deleted = CBool(arsInputFields[14])

'for debug output to the console
PRINT "Adding data for record: " & RecordNum & " ";
PRINT MyContactRecord.FirstName & " " & MyContactRecord.LastName

'all record fields are assigned data, so we add to collection
cTheData.Add(MyContactRecord, CStr(RecordNum))

'for debug output to the console
PRINT cTheData[CStr(RecordNum)].LastName & " stored."

'increment our record counter and clear the sInputLine variable
INC RecordNum
sInputLine = ""
WEND 'loop for more records
'close the file when no more records exist in the file
CLOSE # hFileIn
ELSE 'no file found, put up a message to the user
    Message.Info("No contacts.data file present.", "OK")
ENDIF

'for debug only, we will list all the records in the collection
PRINT "Here is what is stored in cTheData:"
FOR EACH MyContactRecord IN cTheData
    PRINT MyContactRecord.FirstName & " " & MyContactRecord.LastName
NEXT
RETURN cTheData 'return the collection (now filled) with data
END 'of GetData
```

Contact.PutData Method

The reciprocal routine to GetData is PutData, where we will store the collection of data that is considered most current. Whatever edits or changes have been made will be reflected

A Beginner's Guide to Gambas – Revised Edition

in this save operation. We will save the date to a file, storing the record as a comma-delimited string. Each line of output will represent one contact record.

The PutData method is declared as a subroutine since it does not return a variable. It takes a single parameter, the collection object we wish to store. In this case, cTheData is the collection passed into this subroutine.

PUBLIC SUB PutData (cTheData AS Collection)

'this string var will hold each line of data read from the file

DIM sInputLine AS String

'we need a file handle and two string vars for building filenames

DIM hFileOut AS File

DIM filename AS String

DIM fullPath AS String

'this class variable is used to convert the string data into

'a contact record. We declare the variable here and

'will instantiate it below

DIM MyContactRecord AS Contact

'a local integer var to keep track of how many records we added

DIM RecordNum AS Integer

'instantiate a new record that is blank. Note that since we are not

'adding this record to a collection, it is not necessary to renew

'it by re-instantiating it each time.

MyContactRecord = NEW Contact

'start at record 1

RecordNum = 1

'we have a hard-coded file name so we only need the path to the program

'because we will store the data where the program resides.

filename = "contacts.data"

fullpath = Application.Path & / filename

'we have to specify WRITE CREATE when we open the file. If the file

'exists, it will be opened for WRITE. If it does not exist, then we

'will create a new file and write to that.

A Beginner's Guide to Gambas – Revised Edition

OPEN fullpath FOR WRITE CREATE AS #hFileOut

'now, we loop through each record in the collection. Note that we
'use the FOR EACH construct to do this.

FOR EACH MyContactRecord IN cTheData

'we concatenate each field to the last, inserting commas
'to be used as field delimiters for the read operation

```
sInputLine = sInputLine & MyContactRecord.FirstName & ","
sInputLine = sInputLine & MyContactRecord.Initial & ","
sInputLine = sInputLine & MyContactRecord.LastName & ","
sInputLine = sInputLine & MyContactRecord.Suffix & ","
sInputLine = sInputLine & MyContactRecord.Street1 & ","
sInputLine = sInputLine & MyContactRecord.Street2 & ","
sInputLine = sInputLine & MyContactRecord.City & ","
sInputLine = sInputLine & MyContactRecord.State & ","
sInputLine = sInputLine & MyContactRecord.Zip5 & ","
sInputLine = sInputLine & MyContactRecord.Zip4 & ","
sInputLine = sInputLine & MyContactRecord.Areacode & ","
sInputLine = sInputLine & MyContactRecord.Prefix & ","
sInputLine = sInputLine & MyContactRecord.Last4 & ","
sInputLine = sInputLine & MyContactRecord.RecordID & ","
sInputLine = sInputLine & CStr(MyContactRecord.Deleted) & ","
'for debug only, to show what is being stored
PRINT "Storing: " & sInputLine
```

```
'this call is actually what puts the data into the file
PRINT #hFileOut, sInputLine
'clear out our string var to be reused in the next iteration
sInputLine = ""
```

```
NEXT 'iteration of the loop to enumerate objects
'got them all written now, so close the file
CLOSE # hFileOut
RETURN 'without passing a variable back, hence subroutine
END
```

Our program will allow a user to add, change or delete a contact and store that data to a data file. Contact data is managed in our program as a collection of objects that are of type Contact, which we defined in the Contact.class file above. We will allow the user to move to the very first item in the collection, scroll to next or previous items, go to the last item, update data on the current form, save the current data to file, and exit.

All in all, it does most of the basics necessary for contact management. We will also add a search feature (using a module) that will find the first instance of a contact by searching for a last name. Amazingly, to accomplish all of this takes very little code in Gambas!

FMain.class file

Save the Contact.class file and close it. Our contact class definition is complete so now we will need to create a form named FMain. FMain will serve as the main interface between the user and the program and its data. Here is what our finished form will look like:



Figure 82: Finished Contacts program.

Now, let's look at the program that will use the contact class. The program will require four public variables, as shown below. Our constructor routine will automatically look for and load the default contact file, which we named contacts.data. It is not necessary for the user to know or use this filename so it is hard-coded into the methods where it is used. If the constructor routine `_new()` cannot load a file, it will create one dummy record so the user never sees a blank startup screen.

' Gambas FMain class file

'Contacts will represent the collection of records for the addr book

PUBLIC Contacts AS Collection

'represents a single contact record used by all subroutines

PUBLIC ContactRecord AS Contact

'index that points to the current record

PUBLIC RecordPointer AS Integer

A Beginner's Guide to Gambas – Revised Edition

```
'we will need a string var to use when the user searches by last name  
PUBLIC SearchKey AS String
```

FMain Constructor

```
'the constructor routine called when FMain_open() is called  
PUBLIC SUB _new()
```

```
'instantiate the collection that will hold our address records  
Contacts = NEW Collection
```

```
'instantiate a new contact (record) that is blank  
ContactRecord = NEW Contact
```

```
'for debug only  
PRINT "Calling GetData..."
```

```
'call the GetData method defined in the contact class definition  
ContactRecord.GetData(Contacts)
```

```
'set the record pointer to the last record found in the dataset  
RecordPointer = Contacts.Count
```

```
'for debug only  
PRINT "In _new(), rtn fm GetData with: " & RecordPointer & " records."
```

```
'the default constructor will ensure at least one record exists  
'if the Contacts collection's count property shows < 1 record.
```

```
IF RecordPointer < 1 THEN
```

```
    'add our dummy data  
ContactRecord.FirstName = "John"  
ContactRecord.Initial = "Q"  
ContactRecord.LastName = "Public"  
ContactRecord.Suffix = "II"  
ContactRecord.Street1 = "12345 Gambas Drive"  
ContactRecord.Street2 = "Apt 101"  
ContactRecord.City = "Dancer"  
ContactRecord.State = "TX"  
ContactRecord.Zip5 = "77929"  
ContactRecord.Zip4 = "0101"  
ContactRecord.Areacode = "888"
```

A Beginner's Guide to Gambas – Revised Edition

```
ContactRecord.Prefix = "666"
ContactRecord.Last4 = "4444"
ContactRecord.RecordID = 1
ContactRecord.Deleted = FALSE
RecordPointer = ContactRecord.RecordID
Contacts.Add (ContactRecord, CStr(ContactRecord.RecordID) )
ENDIF
```

```
'check our collection to make sure records exist
IF Contacts.Exist(CStr(RecordPointer)) THEN
    'for debug only
    PRINT "Record data exists for " & CStr(RecordPointer) & " records."
    'this for loop is for debug only
    FOR EACH ContactRecord IN Contacts
        PRINT ContactRecord.FirstName & " " & ContactRecord.LastName
    NEXT
ELSE
    'if no records, put up a message box to inform user
    Message.Info("Records do not exist!","Ok")
ENDIF
END
```

Form_Open Subroutine

Our program will automatically open the form using the Form_Open() subroutine below. Since FMain.form is defined as a startup form, it is not necessary to call the FMain.Show method. All we are going to do here is set the caption to display at the top of the window when the program runs and get the record pointed to by the current record pointer. In this case, the pointer should point to the last record loaded from the contacts file after program flow returns from the constructor.

```
PUBLIC SUB Form_Open()
    'set the program caption
    FMain.Caption = " Contacts Manager "

    'start with the record pointed to after the constructor call
    'which should be the last record in the collection

    ContactRecord = Contacts[CStr(RecordPointer)]
    'this subroutine will refresh all fields with current record data
    UpdateForm
END
```

Adding Controls to FMain.Form

It is time to put all of the controls on the form and give them names. We will be using nine ToolButtons, one regular Button, 14 labels and 13 TextBoxes. We will also need to use nine icons, which we can obtain from the `usr/share/icons/default-kde/32x32` folder (*NOTE: your system may be different so browse around to find where the icons are stored if needed*). The icons used for this program are as follows:

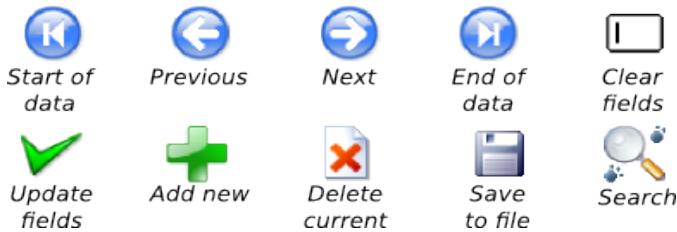


Figure 83: Toolbar icons used in our sample program.

The ToolButtons

The nine ToolButtons that use these icons in their Picture property are named:

```
FirstRecordBtn  PrevRecordBtn  NextRecordBtn  
LastRecordBtn  ClearBtn          Update  
AddBtn         DeleteRecordBtn  SaveAllBtn
```

Create nine ToolButtons and arrange them on the form as shown at the beginning of this chapter. Assign the names shown above to the ToolButtons and set the respective button Picture properties to the names of the icons you chose for your program, based on what your system distribution provides. Worst case scenario is that you can use the built-in icon editor in Gambas to create your own icons.

The Quit Button

Once you have completed the ToolButtons, we will add the regular button as our “Quit” button. Select a Button from the ToolBox and place it on the form in the lower right corner, as close as you can to the Figure above. Set the Text property for this button to “Quit”. As long as we are here, we can code the program's exit routine, which will be called when this button is clicked. Double-click on the Quit button and you will be placed in the code editor. Add this code:

```
PUBLIC SUB QuitBtn_Click()  
'prompt to save data before we quit  
SELECT Message.Question("Save before exit?", "Yes", "No", "Cancel")
```

CASE 1

'if the user says yes, we save

SaveAllBtn_Click

CASE 2 'we don't save, just close

CASE 3 'they canceled, so we return to the program

RETURN

CASE ELSE 'we do nothing but close

END SELECT

ME.Close 'here we close the window

END 'program stops

Adding Label and Textbox Controls

Probably the most tedious part of developing this program is next, adding the Label and TextBox controls to the form. Try to arrange them as shown below. For the Label controls, you can take the default variable names provided by Gambas for all of the labels except the one at the bottom of the form where we will display our *Record n of n* data. That label should be named StatusLabel. The Textbox fields, as shown from top left to bottom right on the form, should be named as follows:

FirstName

MI

LastName

Suffix

Street1

Street2

City

State

Zip5

Zip4

AreaCode

DialPrefix

DialLast4

Hopefully, the names should be self-explanatory (*i.e., self-documenting code*) and you have no problems creating the form. Be careful to use the correct case and type the names exactly as shown above, otherwise you will be tracking down un-declared variables when you try to run the program. At this point, your form, in design mode, should look like this:

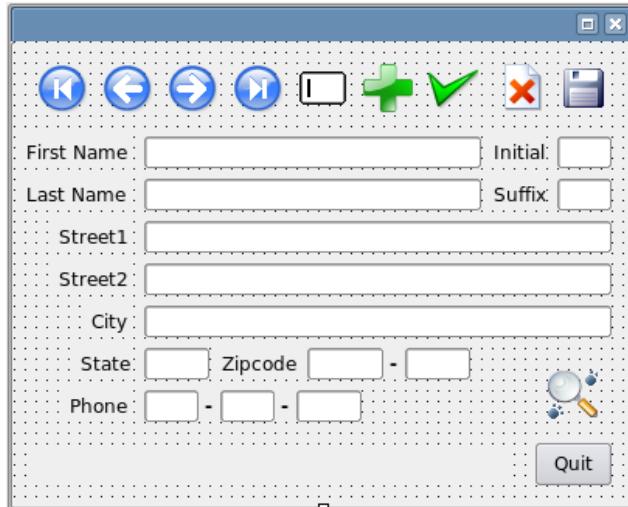


Figure 84: FMain as seen in design mode.

UpdateForm() Subroutine

The UpdateForm() routine will be called every time data on the form has changed and needs to be refreshed so the user will see the updates as they occur. This routine posts each field on the form with the current data found in the collection based on the current record pointer. Also, the label StatusLabel is updated with the current record number of n records in the collection from here. Let's walk through the code:

```
PUBLIC SUB UpdateForm()

'check to ensure there are records before we do anything
IF Contacts.Exist(CStr(RecordPointer)) THEN

    'we found records so now we move to the record at current pointer
    ContactRecord = Contacts[CStr(RecordPointer)]

    'for debug only
    PRINT "In UpdateForm with RecordPointer of:" & RecordPointer

    'assign Textbox.Text values whatever is stored in the contact record
    FirstName.Text = ContactRecord.FirstName
    MI.Text = ContactRecord.Initial
    LastName.Text = ContactRecord.LastName
    Suffix.Text = ContactRecord.Suffix
    Street1.Text = ContactRecord.Street1
    Street2.Text = ContactRecord.Street2
    City.Text = ContactRecord.City
    State.Text = ContactRecord.State
```

A Beginner's Guide to Gambas – Revised Edition

```
Zip5.Text = ContactRecord.Zip5
Zip4.Text = ContactRecord.Zip4
AreaCode.Text = ContactRecord.Areacode
DialPrefix.Text = ContactRecord.Prefix
DialLast4.Text = ContactRecord.Last4

'update the current record status field with current data
StatusLabel.Text = "Record: " & CStr(RecordPointer)
StatusLabel.Text = StatusLabel.Text & " of " & CStr(Contacts.Count)

'call the Refresh method to repaint the form contents
FMain.Refresh

'select the contents of the FirstName field
FirstName.Select

'set the cursor focus to the FirstName field
FirstName.SetFocus
ELSE 'no records exist!
'for debug only
PRINT "In UpdateForm, Record: " & RecordPointer & " not found."
ENDIF
'for debug only
PRINT "Leaving UpdateForm..."
END
```

Coding Toolbuttons: First, Prev, Next, and Last

The first ToolButton we will code is the one that will move us to the beginning of the data. Double-click on the first ToolButton and you should be taken to the code window where the FirstRecordBtn_Click() routine awaits. Add this code:

```
PUBLIC SUB FirstRecordBtn_Click()
'move to the first record
RecordPointer = 1

'retrieve that record from the collection
ContactRecord = Contacts[CStr(RecordPointer)]

'update the form display with the new data
UpdateForm
END
```

A Beginner's Guide to Gambas – Revised Edition

As you can see, moving to the first record was pretty simple. Now, we will code the Prev and Next buttons. Double-click on the second ToolButton (\leftarrow) and you will be taken to the PrevRecordBtn_Click() subroutine. Add this code:

```
PUBLIC SUB PrevRecordBtn_Click()

'for debug only
PRINT "in Prev: current pointer is: " & RecordPointer

'we are moving to a lower record so we decrement the record pointer
DEC RecordPointer

'if we moved past record 1 then reset to record 1
IF RecordPointer = 0 THEN
    RecordPointer = 1
ENDIF

'update our contact variable with the new (prev) record data
ContactRecord = Contacts[CStr(RecordPointer)]

'if we are already at the beginning of the data notify the user
IF RecordPointer = 1 THEN
    'for debug only
    PRINT "At first contact already!"
ELSE
    'for debug only
    PRINT "in Prev - now current pointer is: " & RecordPointer
ENDIF

'update our form with the new data
UpdateForm

END
```

Now, we will code the Next button. Double-click on the third ToolButton (\rightarrow) and you will be taken to the NextRecordBtn_Click() subroutine. Add this code:

```
PUBLIC SUB NextRecordBtn_Click()

'for debug only
PRINT "in Next: current pointer is: " & RecordPointer

'we move forward through the collection so we increment the pointer
INC RecordPointer
```

A Beginner's Guide to Gambas – Revised Edition

```
'if we moved past the last record, reset to the last record
IF RecordPointer > Contacts.Count THEN
    RecordPointer = Contacts.Count
ENDIF

'update the contact record with the new data
ContactRecord = Contacts[CStr(RecordPointer)]

'if we are already at the last record, tell the user
IF RecordPointer = Contacts.Count THEN
    'for debug only
    PRINT "At Last contact already!"
ELSE
    'for debug only
    PRINT "in Next, the current pointer is: " & RecordPointer
ENDIF

'update our form again with the new data
UpdateForm
END 'next record move
```

The last ToolButton we need to code for moving around in the data is for the LastRecordBtn_Click() subroutine. This subroutine will place us at the end of the data collection, at the last record. Double-click on the fourth ToolButton and add this code:

```
PUBLIC SUB LastRecordBtn_Click()
    'set record pointer to last record using collection.count property.
    RecordPointer = Contacts.Count
    'update our record var with the new data for the last record
    ContactRecord = Contacts[CStr(RecordPointer)]
    'update the form with the new data
    UpdateForm
END
```

Coding ToolButtons: Adding a record

Now, let's go through how to add a new contact to the collection. Double-click on the fifth ToolButton, the little person icon, and you will be put in the code window at the AddBtn_Click() subroutine. Let's go through the code:

```
PUBLIC SUB AddBtn_Click()
    'declare a local contact variable just for use in this subroutine
```

A Beginner's Guide to Gambas – Revised Edition

```
DIM MyContactRecord AS Contact
```

```
'instantiate the contact record so we can use it
```

```
MyContactRecord = NEW Contact
```

```
'if there is no first name, we will not add a record
```

```
IF FirstName.Text <> "" THEN
```

```
'for debug only
```

```
PRINT "Adding " & FirstName.Text & " to collection."
```

```
'we are here so we found a first name. Assign the Textbox data
```

```
'to the fields stored in the record:
```

```
MyContactRecord.FirstName = FirstName.Text
```

```
MyContactRecord.LastName = LastName.Text
```

```
MyContactRecord.Initial = MI.Text
```

```
MyContactRecord.Suffix = Suffix.Text
```

```
MyContactRecord.Street1 = Street1.Text
```

```
MyContactRecord.Street2 = Street2.Text
```

```
MyContactRecord.City = City.Text
```

```
MyContactRecord.State = State.Text
```

```
MyContactRecord.Zip5 = Zip5.Text
```

```
MyContactRecord.Zip4 = Zip4.Text
```

```
MyContactRecord.AreaCode = AreaCode.Text
```

```
MyContactRecord.Prefix = DialPrefix.Text
```

```
MyContactRecord.Last4 = DialLast4.Text
```

```
'increment the record key so each key is unique when added to
```

```
'the collection. Gambas requires that this be converted to a
```

```
'string variable when we actually add it below.
```

```
MyContactRecord.RecordID = Contacts.Count + 1
```

```
'if we adding a record it is not tagged as deleted
```

```
MyContactRecord.Deleted = FALSE
```

```
'set global record pointer to the value of the incremented key
```

```
RecordPointer = MyContactRecord.RecordID
```

```
'now, call the Collection.Add method to add the record
```

```
Contacts.Add (MyContactRecord, CStr(RecordPointer) )
```

```
'this is for debug only to show what has been added
```

```
IF Contacts.Exist(CStr(RecordPointer)) THEN
```

```
'for debug only
```

A Beginner's Guide to Gambas – Revised Edition

```
PRINT "Record " & CStr(RecordPointer) & " added!"  
ELSE  
    'for debug only  
    PRINT "Record does not exist!"  
ENDIF  
ELSE 'cannot add a contact without a first name, tell user  
    Message.Info("Cannot add without a first name", "Ok")  
ENDIF  
  
'update the form with the new data  
UpdateForm  
  
'this is for debug purposes, show all records in the collection  
FOR EACH MyContactRecord IN Contacts  
    PRINT "Record " & CStr(RecordPointer) & ":";  
    PRINT MyContactRecord.FirstName & " " & MyContactRecord.LastName  
NEXT  
END 'add routine
```

Coding ToolButtons: Clearing data

The sixth ToolButton is used to clear the data on the form to allow users to start with a fresh, blank form to enter data. It simply blanks out each Textbox field. Double-click on the ClearBtn and enter this code:

```
PUBLIC SUB ClearBtn_Click()  
    FirstName.Text = ""  
    MI.Text = ""  
    LastName.Text = ""  
    Suffix.Text = ""  
    Street1.Text = ""  
    Street2.Text = ""  
    City.Text = ""  
    State.Text = ""  
    Zip5.Text = ""  
    Zip4.Text = ""  
    AreaCode.Text = ""  
    DialPrefix.Text = ""  
    DialLast4.Text = ""  
END
```

Validating User Input

When the user enters data in the Initial field, and also in the Suffix, State, Zip5, Zip4, AreaCode, DialPrefix and DialLast4 fields, we want to ensure that the data fits and they cannot enter extraneous letters or characters. Each of the routines below are activated on a change event. In otherwords, if the text in the TextBox.Text field changes, it will go to the respective _Change() routine below. Each routine works almost identically to limit the number of characters the user can type. If they type more, the extra characters are truncated at the length specified by the program. For example, an area code can be no more than three digits. We could also add code to ensure only digits are entered in numeric fields, etc., but that is not why we are building this program. WI will leave that to you to add as an exercise at the end of this chapter. For each check below, we will need a temporary string var, sTemp. If the length of the TextBox.Text data exceeds the length we allow, we use MID\$ to copy the first n letters to sTemp and then copy that value back into the TextBox.Text property.

```
PUBLIC SUB MI_Change()
DIM sTemp AS String

IF Len(MI.Text) > 1 THEN
    sTemp = Mid$( MI.Text, 1, 1)
    MI.Text = sTemp
ENDIF
END

PUBLIC SUB Suffix_Change()
DIM sTemp AS String

IF Len(Suffix.Text) > 3 THEN
    sTemp = Mid$( Suffix.Text, 1, 3)
    Suffix.Text = sTemp
ENDIF
END

PUBLIC SUB State_Change()
DIM sTemp AS String

IF Len(State.Text) > 2 THEN
    sTemp = Mid$( MI.Text, 1, 2)
    State.Text = sTemp
ENDIF
END

PUBLIC SUB Zip5_Change()
```

A Beginner's Guide to Gambas – Revised Edition

```
DIM sTemp AS String
```

```
IF Len(Zip5.Text) > 5 THEN  
    sTemp = Mid$( M1.Text, 1, 5)  
    Zip5.Text = sTemp  
ENDIF  
END
```

```
PUBLIC SUB Zip4_Change()  
    DIM sTemp AS String
```

```
IF Len(Zip4.Text) > 4 THEN  
    sTemp = Mid$( M1.Text, 1, 4)  
    Zip4.Text = sTemp  
ENDIF  
END
```

```
PUBLIC SUB AreaCode_Change()  
    DIM sTemp AS String
```

```
IF Len(AreaCode.Text) > 3 THEN  
    sTemp = Mid$( AreaCode.Text, 1, 3)  
    AreaCode.Text = sTemp  
ENDIF  
END
```

```
PUBLIC SUB DialPrefix_Change()  
    DIM sTemp AS String
```

```
IF Len(DialPrefix.Text) > 3 THEN  
    sTemp = Mid$( DialPrefix.Text, 1, 3)  
    DialPrefix.Text = sTemp  
ENDIF  
END
```

```
PUBLIC SUB DialLast4_Change()  
    DIM sTemp AS String
```

```
IF Len(DialLast4.Text) > 4 THEN  
    sTemp = Mid$( DialLast4.Text, 1, 4)  
    DialLast4.Text = sTemp  
ENDIF  
END
```

As you can see, lots of repetitive but necessary code. You can go back later and “bulletproof” the input of this program by adding additional validation checks for every possible thing you can think of but I will show you later in this book a better way, building a customized InputBox that will perform the validation process automatically.

Adding a Search Feature

We want the user to be able to find a contact by searching for a last name. There are so many controls on our form now that adding another one would detract from the overall aesthetics and make it appear less than elegant. A better solution is to use a mouse click anywhere on the form itself. It generates an event and will allow us to capture that event and initiate a search process. When this happens, we are going to pop up a customized search form and prompt the user to enter the last name to search for. Here is the code for the mouse down event:

```
PUBLIC SUB Form_MouseDown()
    SearchForm.ShowModal
    Message.Info("Back to main form with: " & SearchKey, "Ok")
    DoFind
END
```

Note that we show the form using the ShowModal method instead of just calling Show. This is because we want the user to enter something and click the Search button. Here is what the form itself looks like in design mode:

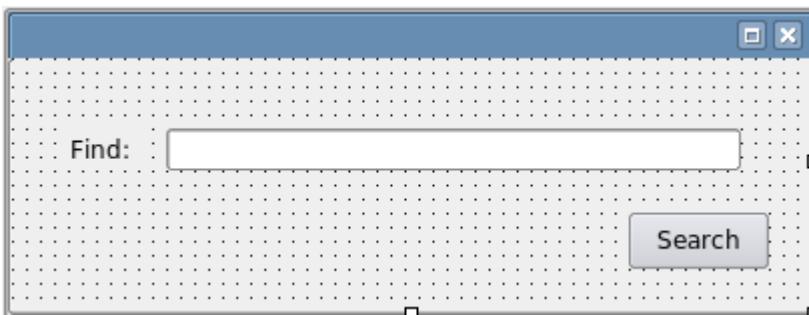


Figure 85: Designing the search form.

Pretty basic, just a label, a TextBox, and a button. From the IDE create a new form, named SearchForm and add the label and button as shown below. Name the button SearchBtn and the TextBox should be named SearchInput. Here is all the code that is needed for the SearchForm:

```
' Gambas SearchForm class file
```

```
PUBLIC SUB SearchBtn_Click()
```

A Beginner's Guide to Gambas – Revised Edition

```
'this is a call to our custom module, search, which has
'a single method, SearchOn, that is explained below
Search.SearchOn( SearchInput.Text)

'for debug only
PRINT "returning: " & SearchInput.Text

'assign the search key to the global var we named in FMain
FMain.SearchKey = SearchInput.Text
'now close our search form
SearchForm.Close
END

PUBLIC SUB Form_Open()
'set the caption for the search window
SearchForm.Caption = " Find by last name "
'highlight and select the SearchInput Textbox
'since there is only one input field, it has focus already
SearchInput.Select
END
```

At this point, you might be wondering what is going on. Basically, in Gambas, public variables are public *to a class file*. When we want to pass parameters *between* classes, we can use the method of passing parameters with functions like we did when we started the program and passed an empty collection variable to the GetData method. Another way to accomplish this is by using modules. The key point to remember is that public variables in modules are public to **all** classes. Essentially, all we have done is create a conduit to pass our search key from the search form back to the main program via the module. Note that the only code needed in the module is a statement that takes the parameter passed from the search form and returns it to the main program. As soon as it comes back from the module, it is assigned to the FMain class public variable SearchKey. Note that we fully qualified the variable name, specifying FMain.SearchKey to ensure it resolves correctly. This is necessary because the SearchButton_Click() subroutine that invoked the module is in a different class file than the FMain class file that called it.

```
' Gambas module file
'PUBLIC SearchKey AS String
PUBLIC FUNCTION SearchOn(sKey AS String) AS String
  RETURN sKey
END
```

A Beginner's Guide to Gambas – Revised Edition

The DoFind Subroutine

At this point, we have popped up the form, obtained the user input of a last name to search for, and have passed that variable back to our program using the module conduit method described above. Now, we are ready to actually search the collection and find the entry. Here is the code to do that:

```
PUBLIC SUB DoFind()
```

```
'declare a contact object for local use
```

```
DIM MyContactRecord AS Contact
```

```
'instantiate it
```

```
MyContactRecord = NEW Contact
```

```
'set the record pointer to the first record in the collection
```

```
RecordPointer = 1
```

```
'for debug only
```

```
PRINT "In DoFind with: " & SearchKey
```

```
'use FOR EACH to iterate through each object in the collection
```

```
FOR EACH MyContactRecord IN Contacts
```

```
'assign each temp record to our global contact object
```

```
ContactRecord = Contacts[CStr(RecordPointer)]
```

```
'if the last name matches search key, update the form and leave
```

```
IF ContactRecord.LastName = SearchKey THEN
```

```
'for debug only
```

```
PRINT "Found: " & MyContactRecord.LastName
```

```
'now update the form
```

```
UpdateForm
```

```
BREAK 'force return with this statement
```

```
ENDIF
```

```
'no match so we increment the record pointer and move to the next
```

```
INC RecordPointer
```

```
NEXT
```

```
END
```

Updating an Existing Record

If the user happens to edit data on an existing record when it is displayed, we want the

data to be saved to the same record. To accomplish this using a collection object is complicated by the fact that each object in the collection must have a unique key. To get around this, we will delete the object with the current key using the built-in method Remove, and re-add the object using the same key but with updated data. Sort of sneaky, but it gets the job done without much effort. Here is how it works:

```
PUBLIC SUB Update_Click()
    'declare a local contact object for us to work with
    DIM MyContactRecord AS Contact
    'instantiate it
    MyContactRecord = NEW Contact
    'now remove the existing record in the collection
    Contacts.Remove(CStr(RecordPointer))
    'then we populate our work record with the form data
    MyContactRecord.FirstName = FirstName.Text
    MyContactRecord.LastName = LastName.Text
    MyContactRecord.Initial = MI.Text
    MyContactRecord.Suffix = Suffix.Text
    MyContactRecord.Street1 = Street1.Text
    MyContactRecord.Street2 = Street2.Text
    MyContactRecord.City = City.Text
    MyContactRecord.State = State.Text
    MyContactRecord.Zip5 = Zip5.Text
    MyContactRecord.Zip4 = Zip4.Text
    MyContactRecord.Areacode = AreaCode.Text
    MyContactRecord.Prefix = DialPrefix.Text
    MyContactRecord.Last4 = DialLast4.Text
    'add the work copy to the collection with the same
    'key that belonged to the record we deleted above
    Contacts.Add (MyContactRecord, CStr(RecordPointer) )

    IF Contacts.Exist(CStr(RecordPointer)) THEN
        'for debug only
        PRINT "Record " & CStr(RecordPointer) & " updated!"
    ELSE
        PRINT "Record not updated!" 'we want this output if failed
    ENDIF
    'update the form with the new data
    UpdateForm
END
```

Deleting a Contact Record

If the user wants to delete a record in a collection we also have to think about how to handle that. Managing which keys are available and which are not could be a programmer's nightmare so an easier approach is to simply mark the record as deleted and blank it. That way, it is available for reuse with the same key. It blanks out the data and, when the save occurs, does not store the old data, effectively deleting it. Only difference is, we can reuse the key by having the user simply enter data in an empty record and click the update button. Here is how this is accomplished:

```
PUBLIC SUB DeleteRecordBtn_Click()
    'declare our local contact object
    DIM MyContactRecord AS Contact
    'instantiate it
    MyContactRecord = NEW Contact
    'remove the current record using the built-in Remove method
    Contacts.Remove(CStr(RecordPointer))
    'clear all the fields on the form
    ClearBtn_Click
    'set the first field with an indicator that the record is deleted
    MyContactRecord.FirstName = "<Deleted Record>"
    'leave the rest blank by simply assigning them after clearing
    MyContactRecord.LastName = LastName.Text
    MyContactRecord.Initial = MI.Text
    MyContactRecord.Suffix = Suffix.Text
    MyContactRecord.Street1 = Street1.Text
    MyContactRecord.Street2 = Street2.Text
    MyContactRecord.City = City.Text
    MyContactRecord.State = State.Text
    MyContactRecord.Zip5 = Zip5.Text
    MyContactRecord.Zip4 = Zip4.Text
    MyContactRecord.Areacode = AreaCode.Text
    MyContactRecord.Prefix = DialPrefix.Text
    MyContactRecord.Last4 = DialLast4.Text
    ' mark record as deleted
    MycontactRecord.Deleted = TRUE
    'add the blanked, deleted record as it exists to the collection
    Contacts.Add (MyContactRecord, CStr(RecordPointer) )
    IF Contacts.Exist(CStr(RecordPointer)) THEN
        'for debugging only
        PRINT "Record " & CStr(RecordPointer) & " marked deleted!"
    ELSE
        'if it fails, we want a message
        Message.Info("Record not marked deleted!", "Ok")
    ENDIF
```

```
'update the form  
UpdateForm  
END
```

Saving Data

The last thing we need to code is the save button. This is really simple because we have already built a method in our Contact class file to do that. Here, all we do is call the PutData method:

```
PUBLIC SUB SaveAllBtn_Click()  
'invoke the PutData method  
ContactRecord.PutData(Contacts)  
'for debug only  
PRINT "Contacts saved to contact.data file."  
END
```

That is all we have left to do. Run the program and enter some data. Be sure to try all the options and create a few records. Save your data to file and reload it. When you are satisfied that it works as we planned, it is ok to go back and comment out all the console output (the PRINT statements) so they will not display at runtime. For all the other programs we have created, we have been content to simply execute them from within the IDE. Since this program can be a useful addition to your desktop, we will now show you how to create a stand-alone executable and allow it to run independent of the Gambas programming environment.

Creating a Stand-alone Executable

It is very easy to create the executable. Simply make sure your program is ready for release (test it first!) and then go to the Project menu and choose Make Executable. The Gambas IDE will create an execution script that will invoke the Gambas interpreter and allow you to execute your code. I created a desktop link to the program and used this for the application path on my system:

```
'/root/My Documents/Gambas for Beginners/Classes/Classes'
```

Of course, **your system will be different**. You will need to point the desktop link to the application to where ever you had the IDE save the executable file. On the Linux distribution I originally used for this program (Linspire 5.0), it would not work until I enclosed the full path with tick marks, shown above. Because the Gambas interpreter is invoked as a system call, the ticks are needed to keep the system from thinking it is passing parameters to **/root/My** (where the first space character occurs). Here is the final result, our program executing in stand-alone mode on the desktop:

A Beginner's Guide to Gambas – Revised Edition

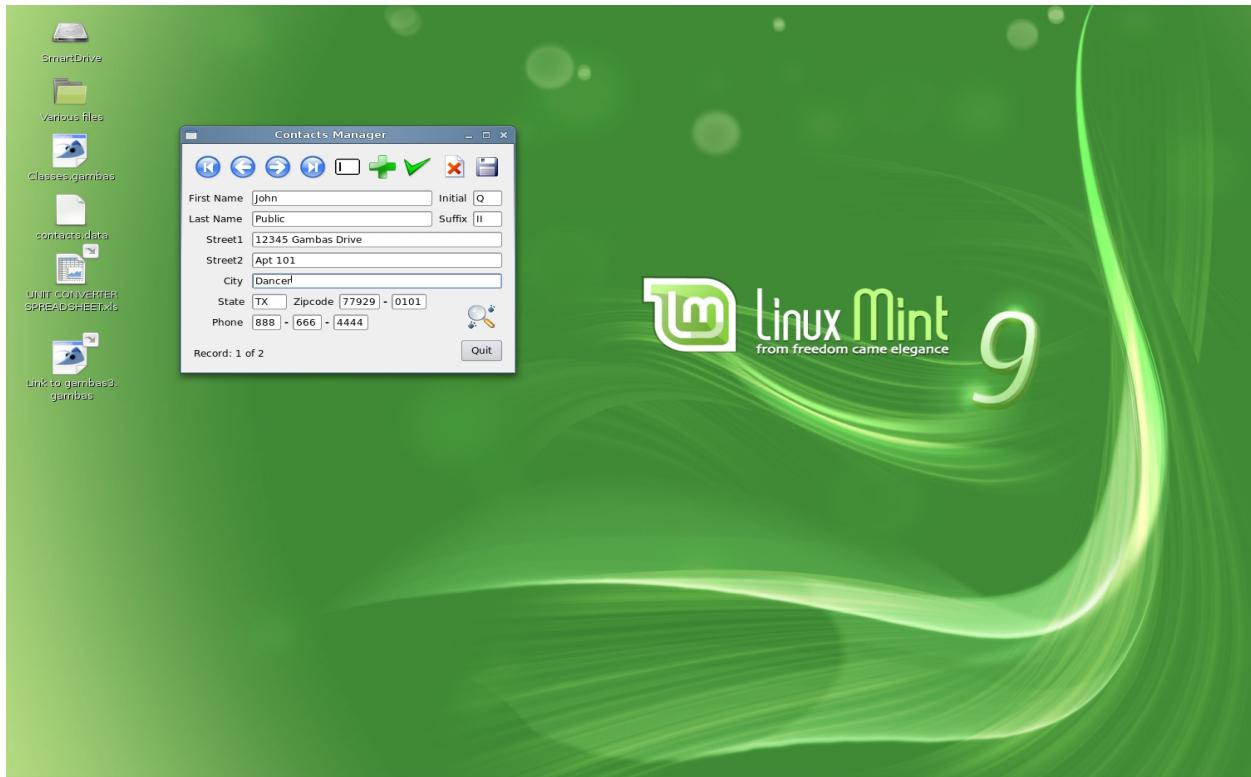


Figure 86: Our finished program running on the Linux Mint desktop.



Chapter 12 – Learning to Draw

Draw is the class used in Gambas for drawing on an object. The object may be of type Picture, Window, Printer, Drawing, or DrawingArea object. This class is static. It is important to remember that before starting to draw anything, you must call the **Begin** method by passing it a handle to the object type you want to draw. Once you do that, you can call any of the drawing methods GB provides to draw points, lines, texts, pictures, etc. However, *it is not clearly documented* that you **must also set** the DrawingArea's **Cached** property to True if you want to see your drawing appear on the form. When your drawing is finished, you must call the **End** method.

Draw Properties

Each item you wish to draw in Gambas is supported with several common attributes, such as background color and foreground color, a fill color, fill style, etc. Some properties in Gambas, such as Clip, are read-only and simply return data to you for use in your code.

Background and Foreground

As we explained in Chapter 4, many of the properties for controls are common among all controls, so we will take the time to explain all of the button properties here, and later in the book we will only explain the properties that are unique to a given control.

Background is defined as [PROPERTY Background AS Integer](#)

Foreground is defined as [PROPERTY Foreground AS Integer](#)

This integer value represents the color used for the current drawing background or foreground color. It is synonymous with the BackColor/ForeColor properties, which will be obsolete in Gambas3. Note that **PROPERTY** is a predefined data-type used internally in Gambas. You can use the GB predefined constants for color to set color value:

Black	Blue	Cyan	DarkBlue
DarkCyan	DarkGray	DarkGreen	DarkMagenta
DarkRed	DarkYellow	Default	Gray
Green	LightGray	Magenta	Orange
Pink	Red	Transparent	Violet
White	Yellow		

To set the Background property to red, you would use this code:

[Draw.Background = Color.Red](#)

Alternatively, if you know the RGB or HSV values for a specific color, GB provides a

means to convert those values to an integer value that can be passed to the Background (*or other color-related*) property. The class Color provides two methods, RGB and HSV that you can use. Here is how those functions are defined:

STATIC FUNCTION RGB (R AS Integer, G AS Integer, B AS Integer [, Alpha AS Integer]) AS Integer

The **RGB** function returns a color value from its red, green and blue components. For the HSV function, use this:

STATIC FUNCTION HSV (Hue AS Integer, Saturation AS Integer, Value AS Integer) AS Integer

HSV returns a color value from its hue, saturation and value components. To use one of these functions to set the button's background color, here is what you could code:

`Draw.Background = Color.RGB(255,255,255)`

This would set the button's background color to white. This is most useful when you are trying to set colors whose values fall outside the default constants values provided in GB.

Clip

This class is static. Clip is a read-only property that returns a virtual object (*.DrawClip*) that is used for managing the clipping area of a drawing. *.DrawClip* is a virtual class used for defining the clipping area of a drawing. The Gambas drawing methods never draw outside the boundaries of the defined clipping area. You cannot use this virtual class as a data-type. The standard calling convention is:

STATIC PROPERTY READ Clip AS .DrawClip

This class can be used as a function. To invoke the function to define a clipping area, use this calling convention:

STATIC SUB .DrawClip (X AS Integer, Y AS Integer, W AS Integer, H AS Integer)

Properties returned by this function include Enabled, H, Height, W, Width, X and Y.

FillColor, FillStyle, FillX, FillY

FillColor returns or sets the color used by filling drawing methods. This class is static. FillStyle returns or sets the style used by filling drawing methods. The Fill class supports several FillStyle predefined constants used to represent fill patterns for drawing:

A Beginner's Guide to Gambas – Revised Edition

BackDiagonal	Cross	CrossDiagonal	Dense12
Dense27	Dense50	Dense6	Dense88
Dense94	Diagonal	Horizontal	None
Solid	Vertical		

Calling convention for FillColor and FillStyle is as follows:

STATIC PROPERTY FillColor AS Integer
STATIC PROPERTY FillStyle AS Integer

FillX and FillY are used to return or set the horizontal/vertical origin of the brushes used by filling drawing methods. Calling convention for FillX and FillY is:

STATIC PROPERTY FillX AS Integer
STATIC PROPERTY FillY AS Integer

Font

Font returns or sets the font used for rendering text on the drawing surface. Calling convention is:

STATIC PROPERTY Font AS Font

To set control font attributes, this code can be used:

```
Draw.Font.Name = "Lucida"  
Draw.Font.Bold = TRUE  
Draw.Font.Italic = FALSE  
Draw.Font.Size = "10"  
Draw.Font.StrikeOut = FALSE  
Draw.Font.Underline = FALSE
```

Invert

Invert is used to stipulate that all drawing primitives will combine their pixel colors with the pixel colors of the destination using an XOR operation. Calling convention is:

STATIC PROPERTY Invert AS Boolean

LineStyle/LineWidth

LineStyle returns or sets the style used for drawing lines. LineWidth returns or sets the width used for drawing lines. The Line class is static and defines the constants used by the Draw.LineStyle property. These constants include:

Dash	Dot	DashDot
DashDotDot	None	Solid

Standard calling convention is:

STATIC PROPERTY LineStyle AS Integer
STATIC PROPERTY LineWidth AS Integer

Transparent

Transparent indicates that some drawing methods like Draw.Text are transparent, i.e., it means they do not fill their background. Calling convention is:

STATIC PROPERTY Transparent AS Boolean

Draw Methods

One of the best ways to learn how to use Gambas drawing methods is to write code that demonstrates the use of each method. In this manner, you can see how to program the method and get immediate gratification by seeing the results. We are going to build a demo application that will show you how to use nearly all of the methods that are supported by the Draw class. We will start by creating a new QT graphical user interface application. Name the project gfxDemo and when the IDE appears, create a form, FMain. Make sure you specify it as a startup class and ensure the controls are public. Here is what the FMain layout will look like at runtime:



Figure 87: gfxDemo FMain layout.

Set the form width to 530 and the height to 490. The default settings for all other properties should be fine for our purposes. Create eight buttons and label them as shown in

the picture above. The button names are TextBtn, InvRectBtn, EllipseBtn, FillRectBtn, PolygonBtn, PolyLineBtn, TileBtn, and QuitBtn. Each button should have a width of 60 and a height of 25. Place them as low on the form as you can without cutting off the bottom of the button. For each button you have created, we will need to create a click event by double-clicking on the button.

As we go through the explanations of each of the following methods, we will be adding code where appropriate. Before we get to the methods themselves, there are a couple of things we need to do with our program. First, we need to add a drawing area control, named **da** to the form. The Drawing Area tool can be found on the Container tab of the Gambas Toolbox. Place it at the top left corner of FMain and make it about 1 inch by 1 inch. We will dynamically resize it in our program. Next, we will use a constructor to start the drawing process, invoking the **Draw.Begin** method and setting the parameters that will be used for our program. Here is the constructor subroutine you need to add:

```
PUBLIC SUB _new()
    'establish width of the drawing area – .W is synonymous with .Width
    da.W = FMain.W - 10
    'establish height of the drawing area – .H is synonymous with .Height
    da.H = FMain.H - 45
    'make the mandatory call to start draw operations
    Draw.Begin(da)
    'set a default line width of 2 pixels
    Draw.LineWidth = 2
    'for the .Clear method to work, the property DrawingArea.Cached must be set to TRUE.
    da.Cached = TRUE
END
```

When the form opens at program start, it will first call the constructor above, then execute the code in the Form_Open subroutine. All we are going to do for this subroutine is set the caption at the top of the form:

```
PUBLIC SUB Form_Open()
    FMain.Text = " Drawing Examples "
END
```

The final routine we will create before exploring the Draw methods is the code needed to stop our program. Double-click on the Quit button and add this code:

```
PUBLIC SUB QuitBtn_Click()
    Draw.End 'close drawing operations
    ME.Close 'close FMain
END
```

Now, we are ready to begin adding code to explore our various drawing tools, the methods of the Draw class.

Text/TextHeight/TextWidth

We will begin with Draw.Text and the two read-only text properties, TextHeight and TextWidth. TextHeight returns the height of a text drawing while TextWidth returns the width of a text drawing. Standard calling convention is:

```
FUNCTION TextHeight ( Text AS String ) AS Integer
```

```
FUNCTION TextWidth ( Text AS String ) AS Integer
```

Double-click on the Text button and add this code:

```
PUBLIC SUB TextButton_Click()
    'this var will be used for our loop counter
    DIM counter AS Integer

    'clear the current drawing area
    da.Clear

    'set foreground color to black
    Draw.Foreground = color.Black

    'set current font to Arial
    draw.Font.Name = "Arial"

    'turn on boldface
    draw.Font.Bold = TRUE

    'turn off italic
    draw.Font.Italic = FALSE

    'turn off underline
    draw.Font.Underline = FALSE

    'set text size to 16 points
    draw.Font.Size = "16"

    'start a loop and continue until we iterate 10 times
    FOR counter = 0 TO 9
        'output text and increment the y position by 40 * counter
```

A Beginner's Guide to Gambas – Revised Edition

```
'the optional width, height parameters are given as 100, 60
```

```
draw.Text("Sample Text", 10, 10 + 40*counter, 100,60)  
NEXT 'iteration of the loop
```

```
'now set font to 24 points  
draw.Font.Size = "24"
```

```
Draw.Foreground = color.Blue 'and change color to blue
```

```
FOR counter = 0 TO 9 'loop another 10 times  
'all we change is x position to move the text right 200 pixels  
draw.Text("More Sample Text", 200, 10 + 40*counter, 100,60)
```

```
NEXT 'iteration
```

```
END ' we are done with text stuff!
```

Once you have added the code above, save your work and click the RUN button. The project will compile and, when completed, here is what you should see:

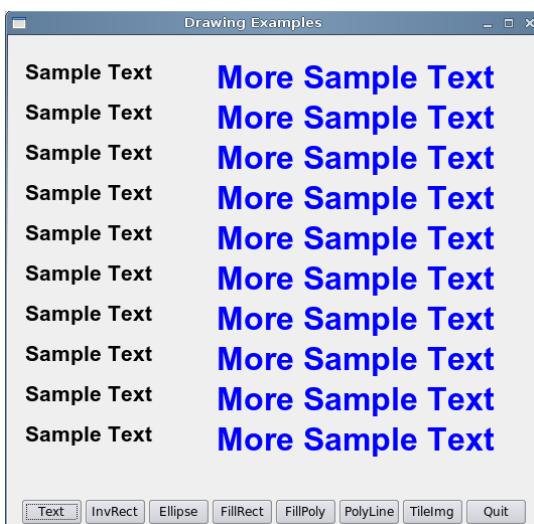


Figure 88: Results of clicking the Text Button.

Draw Primitives: Point/Rect/Ellipse/Line

The Draw.Point method draws a single pixel while Draw.Rect draws a rectangle. Calling convention for these two methods is:

```
STATIC SUB Point (X AS Integer, Y AS Integer )
```

STATIC SUB Rect (X AS Integer,Y AS Integer,Width AS Integer,Height AS Integer)

We are going to use the InvRect button to illustrate how these methods are used. Double-click on the InvRect button and enter the following code:

```
PUBLIC SUB InvRectBtn_Click()
    'use this var for our loop counter
    DIM counter AS Integer
    'clear the drawing area
    da.Clear
    'make sure fill style is set to None
    draw.FillStyle = fill.None
    'set color to cyan
    draw.Foreground = color.cyan
    'this will make the cyan rectangle appear red
    draw.Invert = TRUE

    FOR counter = 0 TO 15 'loop 16 times
        'draw a rect, inc the startx,y and end x,y positions 5* counter val
        Draw.Rect(5+5*counter, 5+5*counter, da.W/2+5*counter, da.H/2 + 5*counter)
        'if we hit an even number, invoke toggle Invert property on/off
        IF counter MOD 2 = 0 THEN
            draw.Invert = FALSE
        ELSE
            draw.Invert = TRUE
        ENDIF
    NEXT 'iteration of the loop
    'ensure invert is set to off
    draw.Invert = FALSE
    'set current fgd color to black
    draw.Forecolor = color.Black
    'set a point midscreen
    draw.Point(da.W/2, da.H/2)
    'now we will make a tiny crosshair from the first point by moving one pixel up, down, left and right of the center
    'dot we set with the first draw.point call
    draw.Point(da.W/2-1, da.H/2)
    draw.Point(da.W/2+1, da.H/2)
    draw.Point(da.W/2, da.H/2-1)
    draw.Point(da.W/2, da.H/2+1)
END 'of the rect/point demo code
```

Save your work and run the program. When you click the InvRect button, you should see something similar to this:

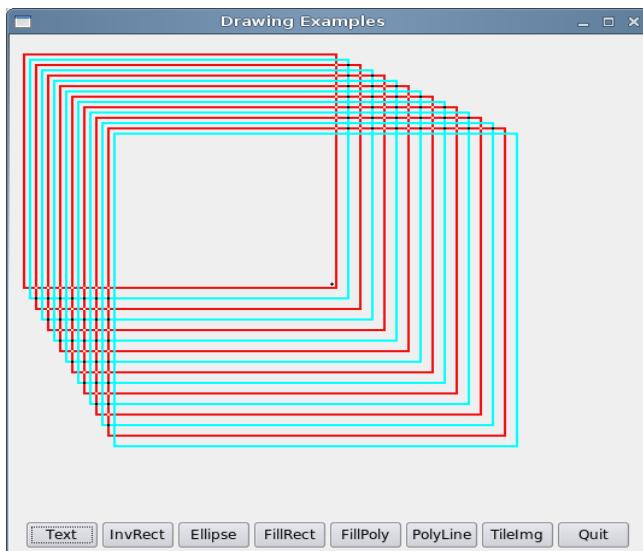


Figure 89: Results of InvRect button click. Note the tiny black crosshair center screen.

The Draw.Line method draws a line. The `x1, y1` points represent the start vertex of the line, `x2, y2` represent the end vertex of the line. The line method uses properties from the Line class to establish line style and thickness. The constants used by the Draw.LineStyle property are *Dash*, *DashDot*, *DashDotDot*, *Dot*, *None*, and *Solid*. Line thickness is set using the Draw.LineWidth property and width is set in pixels. The calling convention for this method is:

```
STATIC SUB Line(X1 AS Integer,Y1 AS Integer,X2 AS Integer,Y2 AS Integer)
```

Draw.Circle will draw a circle. The calling convention for Draw.Circle is:

```
STATIC SUB Circle ( X AS Integer, Y AS Integer, Radius AS Integer [ , Start AS Float, End AS Float ] )
```

X and Y represent the center point of the circle. If you specify the optional parameters of start and length, these represent starting angles (in radians) where the drawing will start and stop.

Draw.Ellipse will draw an ellipse or circle. Remember, a circle is an ellipse with equal width and height parameters. The calling convention for Draw.Ellipse is:

```
STATIC SUB Ellipse ( X AS Integer, Y AS Integer, Width AS Integer, Height AS Integer [ , Start AS Float, Length AS Float ] )
```

X and Y represent the center point of the circle or ellipse. Width can be thought of as the diameter along the horizontal plane, height is the diameter along the vertical plane. We will illustrate this in our example below using the Ellipses button. Double-click on the Ellipses button and enter this code:

```
PUBLIC SUB EllipseBtn_Click()
'declare some variables to work with
DIM x1 AS Integer
DIM x2 AS Integer

'we will need a counter variable
DIM i AS Integer

'and a var to hold the value of line width as we change it
DIM lwidth AS Integer

DIM iRadius AS Integer
DIM iWidth AS Integer
DIM iHeight AS Integer

'We will need to convert the degree values of i and i + 30 to Radians
DIM sAngle1 AS Float
DIM sAngle2 AS Float

'clear our display before we begin
da.Clear

'set initial vectors
iRadius = 50
iWidth = 100
iHeight = 100

'change the color to dark blue
Draw.Foreground = color.DarkBlue

'set up for a solid fill. Any fillable container that is called after
'this call will be filled with the pattern specified below
draw.FillStyle = fill.Solid

'set up a loop that will create 12 increments
FOR i = 0 TO 360 STEP 30
    'if we fall on a 90 degree increment, switch colors
    IF i MOD 45 = 0 THEN
        draw.FillColor = color.Yellow
    ELSE
        draw.FillColor = color.Red
    ENDIF
```

A Beginner's Guide to Gambas – Revised Edition

```
sAngle1 = Rad(i)
sAngle2 = Rad(i + 30)
```

'draw a filled circle of 12 segments, starting at each value of i and sweeping for 30 degrees, as specified in the optional params

```
Draw.Circle(60, 60, iRadius, sAngle1, sAngle2)
```

'Locate the center at pixels x = 60, y = 60.

'The next line, uses the .Ellipse method to make a circle, with
'the width and height being equal. Because the width and height
'of 100 pixels has the same effect as a radius of 50, .Ellipse
'produces the same result as the .Circle method.

```
Draw.Ellipse(90, 90, iWidth, iHeight, sAngle1, sAngle2)
```

'Locate the center at pixels x = 90, y = 90.

```
NEXT
```

'now, turn off the fill style

```
draw.FillStyle = fill.None
```

'set our fgd color to black

```
draw.Foreground = color.Black
```

'start with a line width of a single pixel

```
lwidth = 1
```

'loop 8 times, moving down along the y axis in 25 pix increments

```
FOR i = 10 TO 200 STEP 25
```

'draw a line

```
draw.Line(da.W/2, i, da.W/2 + 150, i)
```

'increase line thickness variable

```
INC lwidth
```

'set the new width

```
draw.LineWidth = lwidth
```

```
NEXT 'iteration of the loop
```

'out of the loop, reset to a default of 2 pix wide

```
draw.LineWidth = 2
```

'for the ellipses below, set x at midscreen horiz axis - 25 pixels

```
x1 = da.W/2-25
```

```
'set our y at midscreen on the vertical axis
yl = da.H/2

'now, lets start another loop and draw some ellipses
FOR i = 1 TO 40 STEP 3 'lets make 13 of them

'moving this one left along the horiz axis
Draw.Ellipse(xl-i*5, yl, 75, 200)

'and moving this one right along the horiz axis
Draw.Ellipse(xl+i*5, yl, 75, 200)

'if we hit an even number in our loop change colors
IF i MOD 2 = 0 THEN
    draw.Foreground = color.Red
ELSE
    draw.Foreground = color.Black
ENDIF
NEXT 'iteration of the loop

END 'end of the ellipses and lines demo code
```

Save your work and run the program. When you click the Ellipses button, you should see something similar to this:

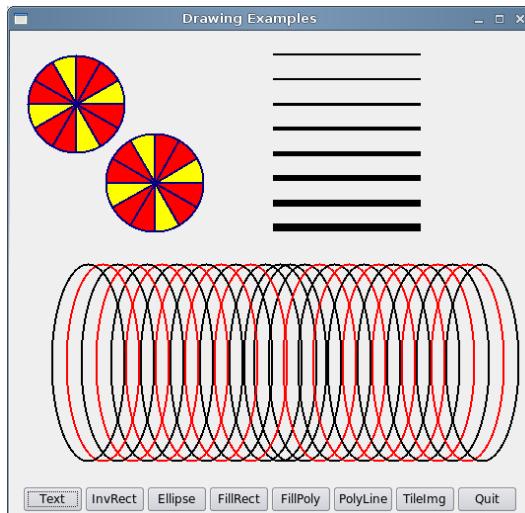


Figure 90: Using the Ellipses button to demonstrate drawing lines and ellipses.

A Beginner's Guide to Gambas – Revised Edition

Our next code segment will use the FillRect button. We are re-visiting the Draw.Rect method to show you how to use the Draw.FillStyle to create filled objects. Create a click-event and enter the following code:

```
PUBLIC SUB FillRectBtn_Click()
    'create some local vars to work with
    DIM xl AS Integer
    DIM x2 AS Integer
    DIM yl AS Integer
    DIM y2 AS Integer

    'clear the drawing area
    da.Clear

    'set the rect top left corner
    xl = 20
    yl = 80

    'set the rect bot right corner
    x2 = xl+50
    y2 = yl+50

    'set fgd color to red
    draw.FillColor = color.Red

    'specify a horiz style fill pattern
    draw.FillStyle = fill.Horizontal

    'draw a rect, if a fill pattern is specified, it is autofilled
    Draw.Rect(xl,yl,x2, y2)

    'set the rect top left corner
    xl = 220
    yl = 180

    'set the rect bot right corner
    x2 = xl+50
    y2 = yl+50

    'set fgd color to blue
    draw.FillColor = color.Blue
    'specify a cross style fill pattern
    draw.FillStyle = fill.Cross
```

```
'draw a rect, if a fill pattern is specified, it is autofilled
Draw.Rect(x1,y1,x2, y2)
'turn off the fill
draw.FillStyle = fill.None
END
```

Save your work and run the program. When you click the FillRect button, you should see something similar to this:

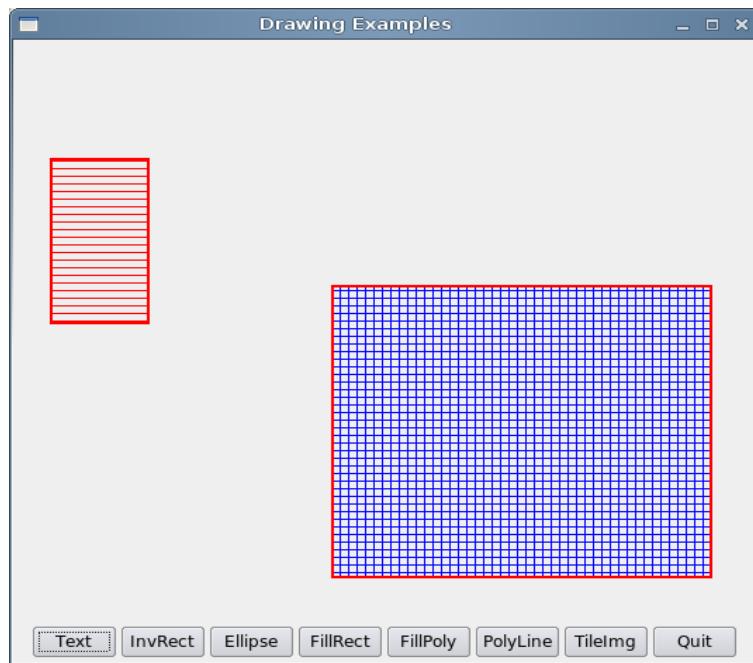


Figure 91: Output after clicking the FillRect button.

Note: in order to graphically display data in a program, you've seen how you could create a circle and fill it in order to generate pie charts. If you wanted to display a rectangle with grid-lines for a line chart, bar chart, etc., you could draw a rectangle and use a FOR...NEXT loop to draw the individual lines of the grid inside the rectangle, using STEP to control the spaces between lines according to your needs.

Draw Primitives: Polygon and Polyline

The Draw.Polygon method draws a polygon with n vertices. The vertices are specified in an Integer array using this calling convention:

STATIC SUB **Polygon (Points AS Integer[])**

The Points parameter is an array of Integer containing the coordinates of the polygon

vertexes. The coordinates are assumed to be in x,y format so there must be two integers in the array for each vertex of the polygon.

The Draw.Polyline works nearly the same as the Polygon routine except that with a polyline the figure being drawn does not need to be closed. This is sometimes referred to as a void polygon. A polyline is a series of vertices joined from one to the next by a line. The line will draw from the last vertex to the next vertex specified in the array. The calling convention for Polyline is:

STATIC SUB Polyline (Points AS Integer[])

The Points parameter is an array of Integer containing the coordinates of the polygon vertexes. So there must be two integers in the array for each vertex. We will use the FillPoly button on our FMain to demonstrate the use of the Draw.Polygon method. Double-click on the FillPoly button to create a click event and when the code window appears, enter this code:

PUBLIC SUB PolygonBtn_Click()

```
'declare some work variables to create a polygon
DIM xl AS Integer
DIM yl AS Integer
DIM x2 AS Integer
DIM y2 AS Integer
DIM x3 AS Integer
DIM y3 AS Integer

'we are going to create a triangle for our polygon
'and store the vertices in the integer array triangle
DIM triangle AS Integer[]

'clear the display area
da.Clear

'set the vertices for each leg of the triangle
'starting with the top part of an equilateral triangle
xl = da.w/2 ' set the horiz axis to mid-screen
yl = 10 'set vertical axis to top + 10 pixels

'now, we will do the left leg of the triangle
'starting on the extreme left edge plus 10 pixels
x2 = da.Left + 10
'and setting our y position to drawingarea height - 200 pixels
y2 = da.H - 200
'the right leg sets horiz position far right - 20 pixels in
x3 = da.W - 20
```

A Beginner's Guide to Gambas – Revised Edition

```
'and the vert axis will be same as the second leg  
y3 = y2  
  
'all vertices defined, load the integer arrays with the triangle  
triangle.Add(x1, 0)  
triangle.Add(y1, 1)  
triangle.Add(x2, 2)  
triangle.Add(y2, 3)  
triangle.Add(x3, 4)  
triangle.Add(y3, 5)  
  
'set a fill style  
draw.FillStyle = fill.Dense63  
'and a fill color  
draw.FillColor = color.DarkMagenta  
'and draw the polygon using the triangle array data  
draw.Polygon(triangle)  
'remember to set fill style to None  
draw.FillStyle = fill.None  
'call refresh to see our work  
da.Refresh  
END ' and we are done with the polygon routine
```

Save your work and run the program. When you click the FillPoly button, you should see something similar to this:

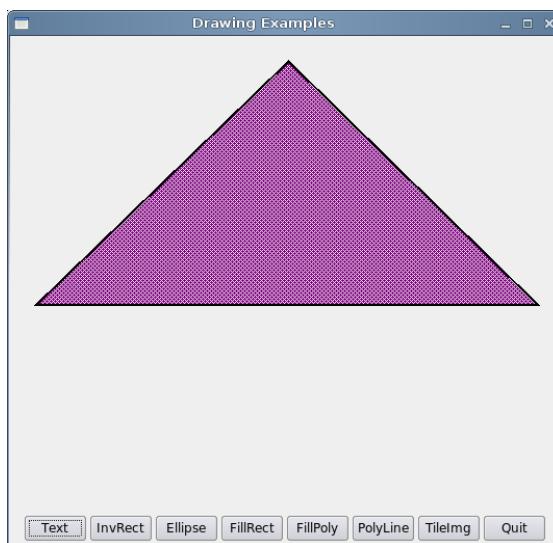


Figure 92: Using Draw.Polygon to draw a triangle.

A Beginner's Guide to Gambas – Revised Edition

Now, let's see how the Draw.Polyline method works. Double-click on the Polyline button and create a click event. When the code window appears, enter this code:

```
PUBLIC SUB PolyLineBtn_Click()
'declare some variables to create our vertices
DIM x1 AS Integer
DIM y1 AS Integer
DIM x2 AS Integer
DIM y2 AS Integer
DIM x3 AS Integer
DIM y3 AS Integer
DIM x4 AS Integer
DIM y4 AS Integer
'declare an integer array to hold the polyline data
DIM lines AS NEW Integer[ ]
'clear the drawingarea
da.Clear
'start our first point midscreen about 10 pixels from the top
x1 = da.w/2
y1 = 10
'the next point will be 10 pixels in from far left, down 200 pixels
x2 = da.Left + 10
y2 = da.H - 200
'our 3rd vertex will be dead-center of the drawing area on horiz axis
'and very bottom of the drawing area for vert axis
x3 = da.W - 20
y3 = da.H - 20
'and the 4th vertex will be dead-center x and y of the drawing area
x4 = da.W/2
y4 = da.H/2
'load the vertices into the array; if we wanted to close the polygon, all we need to do is add the
'x1,y1 vertices to the end of the array. Just uncomment the last two array assignments to try it and see
lines.Add(x1, 0)
lines.Add(y1, 1)
lines.Add(x2, 2)
lines.Add(y2, 3)
lines.Add(x3, 4)
lines.Add(y3, 5)
lines.Add(x4, 6)
lines.Add(y4, 7)
'lines.Add(x1, 8)
'lines.Add(y1, 9)
'ensure any fill styles are turned off
```

```

draw.FillStyle = fill.None
'set the color to dark green
draw.Foreground = color.DarkGreen
'draw the lines
draw.Polyline(lines)
END 'of the polyline demo routine

```

Save your work and run the program. When you click the Polyline button, you should see something similar to this:

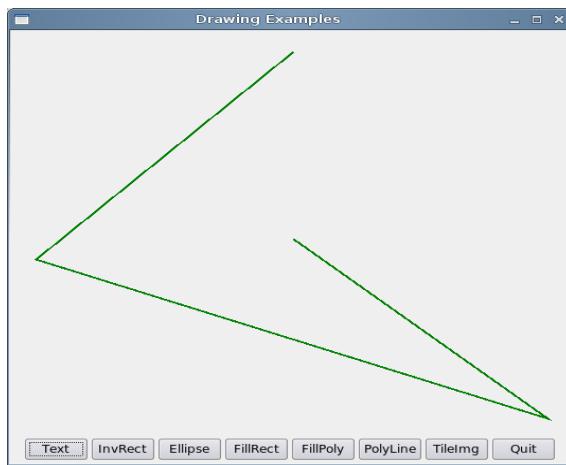


Figure 93: Using Draw.Polyline to draw lines.

Image/Picture/Tile

The Image class draws an Image, or part of it. The image contents are stored in the process memory, not in the display server like a Picture. This class is creatable. If Width and Height are not specified, then the new image is void. Calling convention is:

```
STATIC SUB Image ( Image AS Image, X AS Integer, Y AS Integer [ , SrcX AS Integer, SrcY AS Integer, SrcWidth AS Integer, SrcHeight AS Integer ] )
```

The following code would create a new image:

```

DIM hImage AS Image
hImage = NEW Image ( [ Width AS Integer, Height AS Integer ] )

```

This class acts like an array of pixels. Think of X as rows and Y as columns of image pixels. Here is how the code to get or set a pixel of an image would work:

```

DIM hImage AS Image
DIM anInteger AS Integer

```

anInteger = hImage [X AS Integer, Y AS Integer]

The code above will return the color of an image pixel at X, Y while the following code will set the color of an image pixel to the value of anInteger:

hImage [X AS Integer, Y AS Integer] = anInteger

Image allows you to use the following properties: Depth, Height, Picture, and Width.

Depth returns or sets the depth of the image. When setting depth values, you must only use the following values: 2, 8, 16, and 24. It is declared as:

PROPERTY Depth AS Integer

Height is a read-only attribute that returns the height of the image as an integer. It is declared as:

PROPERTY READ Height AS Integer

Width is a read-only attribute that returns the width in pixels of the image. It is declared as:

PROPERTY READ Width AS Integer

Picture (read-only) converts the image into a picture, and returns it. It is declared as:

PROPERTY READ Picture AS Picture

The Image class supports several methods. These include: Clear, Copy, Fill, Flip, Load, Mirror, Replace, Resize, Rotate, Save, and Stretch.

Clear will clear the image, setting all pixel values to zero.

Copy returns a copy of the image, or a copy of a part of it. The copy method is declared as a function, as shown below:

FUNCTION Copy ([X AS Integer, Y AS Integer, Width AS Integer, Height AS Integer]) AS Image

Fill fills the image with a specified color. Calling convention is:

SUB Fill (Color AS Integer)

Flip returns a mirrored copy of the image. The mirrored copy is created using the horizontal axis of the image.

FUNCTION Flip () AS Image

Load and Save load or saves an image from a file or to a file respectively. The file extension of *Path* will specify the graphics file format used with the saved image. Currently supported file formats are JPEG, PNG, BMP, GIF and XPM. Calling convention is:

SUB Load (Path AS String)

SUB Save (Path AS String)

Mirror returns a mirrored copy of the image. The mirrored copy is created using the vertical axis of the image.

FUNCTION Mirror () AS Image

Replace swaps an existing *OldColor* value with the color specified by *NewColor*, as shown below:

SUB Replace (OldColor AS Integer, NewColor AS Integer)

Resize Resizes the image to the size specified by the width and height parameter given:

SUB Resize (Width AS Integer, Height AS Integer)

Rotate returns a rotated copy of the image. Calling convention is as follows:

FUNCTION Rotate (Angle AS Float) AS Image

The function will rotate the image *n* radians. The angle is specified in radians, so if you want to use degrees for the value of Angle, state the argument as (Rad(Angle) AS Float).

Stretch returns a stretched copy of the image. If the optional parameter Smooth is specified as True, a smoothing algorithm is applied to the returned result. The calling convention is:

FUNCTION Stretch (Width AS Integer, Height AS Integer [, Smooth AS Boolean]) AS Image

The Picture class draws a picture, or part of a picture. The picture contents are stored in the display server, not in the process memory like an Image. Each picture can have a transparency mask. This feature can be set explicitly at picture instantiation, or implicitly when loading an image file that has transparency like PNG. When drawing on a picture having a mask, the picture *and the mask* are modified accordingly. Calling convention is:

STATIC SUB Picture (Picture AS Picture, X AS Integer, Y AS Integer [, SrcX AS Integer, SrcY AS Integer, SrcWidth AS Integer, SrcHeight AS Integer])

This class is creatable. The following code shows you how to create a Picture:

DIM hPicture AS Picture

hPicture = NEW Picture ([Width AS Integer, Height AS Integer, Transparent AS Boolean])

If the optional Width and Height parameters are not specified, the new picture is void. You can specify if the picture has a mask with the Transparent parameter. This class acts like an array.

DIM hPicture AS Picture

hPicture = Picture [Path AS String]

The code above will return a picture object from the internal picture cache. If the picture is not present in the cache, it is automatically loaded from the specified file. In order to insert a picture in the internal picture cache, use this code:

DIM hPicture AS Picture

Picture [Path AS String] = hPicture

Depth returns or sets the depth of the picture. When setting depth values, you must only use the following values: 2, 8, 16, and 24. It is declared as:

PROPERTY Depth AS Integer

Height is a read-only attribute that returns the height of the picture as an integer. It is declared as:

PROPERTY READ Height AS Integer

Width is a read-only attribute that returns the width in pixels of the picture. It is declared as:

PROPERTY READ Width AS Integer

Image is a read-only property that converts the picture into an image and returns it. It is declared as:

PROPERTY READ Image AS Image

We will use an example from the Gambas Wiki to show you how this works. We will show you how to take a screenshot with code and save the image. The image of the desktop is taken as a picture object when the button on the form is clicked. This picture object is subsequently converted into an image object and displayed as an image in a drawingarea.

Start Gambas and create a new QT graphical user interface project, named gfxDemo2. In “Options”, click on “Internationalization”. When the IDE appears, create a startup class

A Beginner's Guide to Gambas – Revised Edition

form named FMain. From the properties window for FMain, set the AutoResize property to True so you can stretch the form to see the image when it is added. Add a button named Button1 and a DrawingArea named DrawingArea1. Place the DrawingArea at the top left corner of the form and make it about one inch wide and one inch high. Now, put the button at the bottom right corner of the form. Add the caption “*Snapshot*” to the button. Double-click on the button to create a click event and add this code:

```
' Gambas class file

PUBLIC SUB Button1_Click()
'declare a var for our picture
p AS NEW Picture

'and one for our image
i AS NEW Image

'Hide the current control so it is not seen in the screenshot
ME.Visible = FALSE      'ME returns a reference to the current object.
' Give the current form a chance to hide
WAIT 0.5
'Get the picture object from the desktop using .Grab method.
p = Desktop.Grab()
'Restore visibility of the control.
ME.Visible = TRUE

'assign the picture using the image property to
'convert screenshot to an image
i = p.image

'specify that the contents of the drawing area are cached in memory
DrawingArea1.Cached = TRUE

'resize drawing area to size of the image converted from picture
DrawingArea1.Resize(i.Width,i.Height)

'clear the drawing area
DrawingArea1.Clear()

'call the Draw.Begin (mandatory) method to start to draw
Draw.Begin(DrawingArea1)

'put the image in the drawing area starting at origin 0,0
Draw.Image(i, 0, 0)
```

```
'call the (mandatory) Draw.End method to quit drawing
Draw.End

'make the image visible
DrawingArea1.Visible = TRUE

'refresh the drawing area to see what was done
DrawingArea1.Refresh
'NOTE: Using the Picture.Save method, you could write code to save the
'screenshot to disk.
END
```

Save your work and run the program. Click the button and a screenshot of your desktop will be placed on the drawing area of the form. Pretty easy using the Gambas methods, isn't it?

Note: In Gambas 2, the Desktop class has the static method .Grab, from the Qt3 library. The screenshot area can be changed by dragging the borders, but the origin of the screenshot will always be pixel 0,0 – the upper left-hand corner of the screen. Gambas 3 uses the updated Qt4 library, and .Grab has been replaced with the method .Screenshot. The calling convention for .Screenshot is:

STATIC FUNCTION Screenshot ([X AS Integer, Y AS Integer, Width AS Integer, Height AS Integer]) AS Picture

Like .Grab, .Screenshot returns a screenshot of the desktop as a Picture object. If the X, Y, Width, Height arguments are specified, they define the part of the screen that will be returned. If they are not defined, the area of the desktop covered by the program window is returned (again, starting at 0,0). In Gambas 3, the line p = Desktop.Grab() can be replaced with something much more sophisticated: p = Desktop.Screenshot(x, y, width, height). With this syntax, you can write code to interactively let the user define the Grab area in pixels, so you no longer are limited to the origin of the screenshot at pixel 0,0.

Transparent is a Boolean flag that can be read or set. It indicates whether or not the picture has a mask. Calling convention is:

PROPERTY Transparent AS Boolean

Methods supported by Picture include Clear, Copy, Fill, Flush, Load, Resize, and Save.

Clear will clear the picture, setting all pixel values to zero.

Copy returns a copy of the picture, or a copy of a part of it. The copy method is declared as a function, as shown below:

FUNCTION Copy ([X AS Integer, Y AS Integer, Width AS Integer, Height AS Integer]) AS Image

Fill fills the picture with a specified color. Calling convention is:

SUB Fill (Color AS Integer)

Flush Flushes the internal picture cache. The method takes no parameters and is called using this convention:

STATIC SUB Flush ()

Load and Save loads or saves a picture from a file or to a file respectively. The file extension of *Path* will specify the graphics file format used with the saved picture. Currently supported file formats are JPEG, PNG, BMP, GIF and XPM. Calling convention is:

SUB Load (Path AS String)

SUB Save (Path AS String)

Resize resizes the image to the size specified by the width and height parameter given:

SUB Resize (Width AS Integer, Height AS Integer)

The **Tile** method draws a tiled picture in the DrawingArea. Calling convention is as follows:

STATIC SUB Tile (Picture AS Picture, X AS Integer, Y AS Integer, Width AS Integer, Height AS Integer)

Let's now go back to our original gfxDemo program and finish coding the last button. Start Gambas and load the gfxDemo project. Open FMain and double-click on the TileImg button. When the code window appears, add this code:

PUBLIC SUB TileBtn_Click()

'declare our local vars here

'we will need a picture var to hold the picture we load

DIM mypic AS Picture

'we need a counter var for our loop work

DIM counter AS Integer

'we are also going to need some integer vars

DIM i AS Integer

DIM j AS Integer

'set i and j to zero to start

```
i = 0
j = 0

'Instantiate picture variable
mypic = NEW Picture

'clear the drawing area
da.Clear

'now load the picture using the load method
mypic = mypic.Load("gambas3.png")

'we will tile the picture i columns across per j rows down
'in our case we will have 4 rows of 3 images per row
'so we will need to loop 12 times

FOR counter = 0 TO 11

'draw the tile

draw.Tile(mypic, (i*175)+20, j, 170,105) "Adding 20 to the x value
'just improves left-to-right centering for this specific image.
'increment our column counter

INC i

'check to see if we have three across

IF i MOD 3 = 0 THEN

  'if so, move value of j (our y axis) image.height + 5 pixels down

  j = j + 110

  'reset column counter to zero

  i = 0

ENDIF 'our check to see if 3 across

NEXT 'iteration of the loop

END 'of the TileImg button code
```

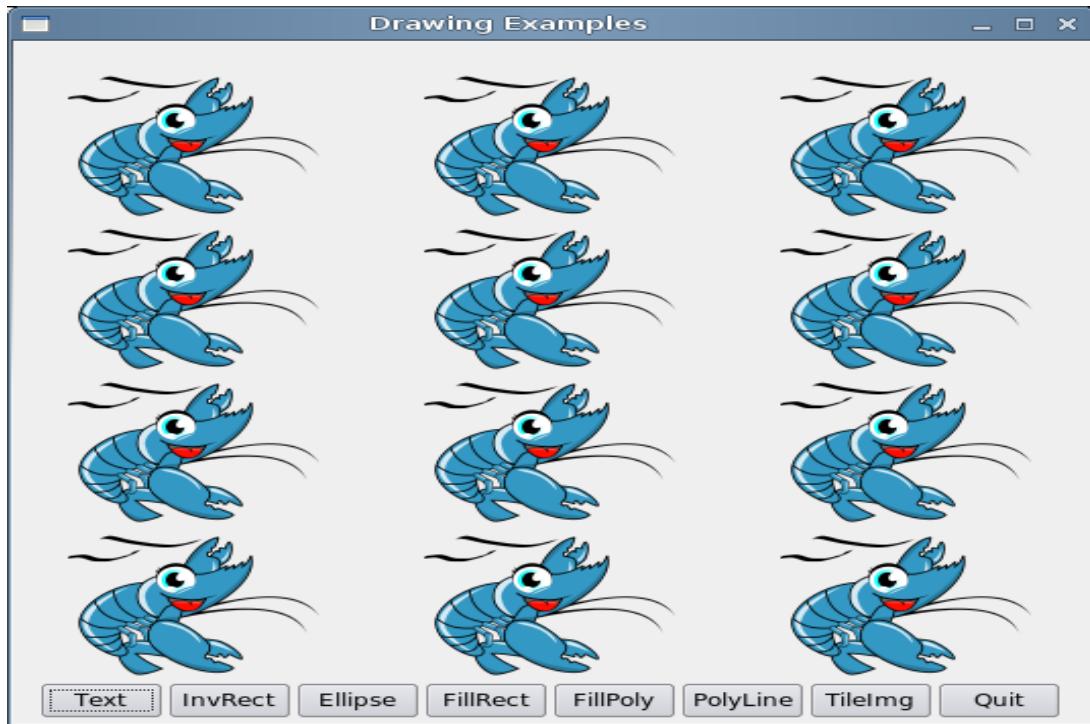


Figure 94: Using tiled images as a background on a form.

The last thing we need to do is get an image to use for our program. For our program, I have chosen the Gambas mascot, which we used from an earlier project. You can copy this image from our FirstProject. From the project window, on the TreeView Data folder, select New Image.

When the window appears, you can select the Existing tab and browse to the folder FirstProject (assuming you named your projects as we specified in the book) and choose the “gambas3.png” image, or whatever the filename is that Gambas is currently using.

Save it to the gfxDemo project as “*Gambas shrimp.png*” and that is all there is to it. If, for some reason, you cannot find the image on your system, simply go to the Gambas web site and use the right mouse button over the mascot image to save a copy to our gfxDemo project folder.

Check the width and height of your image in the file properties, so that you can use that information in the draw.Tile arguments. Now, save your work and run the program. Click the TileImg button and you should see something similar to this:

Drawing with a Drawing object

In Gambas, a drawing object is based on the use of Scalable Vector Graphics, or SVG.

According to the SVG specification¹⁵, “*SVG is a language for describing two-dimensional graphics in XML. SVG allows for three types of graphic objects: vector graphic shapes (e.g., paths consisting of straight lines and curves), images and text. Graphical objects can be grouped, styled, transformed and composited into previously rendered objects. The feature set includes nested transformations, clipping paths, alpha masks, filter effects and template objects. SVG drawings can be interactive and dynamic. Animations can be defined and triggered either declaratively (i.e., by embedding SVG animation elements in SVG content) or via scripting.*”

Inkscape¹⁶ is a popular program used in the Linux community for creating SVG images and is what this author has chosen to use with this book since it is freely available on the Internet. Inkscape is an open source vector graphics editor, with capabilities similar to Illustrator, Freehand, CorelDraw, or Xara X using the W3C standard Scalable Vector Graphics (SVG) file format.

NOTE: As of this writing, *Gambas2 does not properly handle SVG drawings, despite some (very limited) documentation that you may find on the Web. However, the authors have evaluated some basic graphics functions in the Gambas3 Alpha version and Gambas3 does seem to handle SVG correctly, as well as introducing a number of sophisticated graphics functions not previously available. In Gambas3, using the SvgImage class, you can load, paint, save, or clear an SVG image. The properties that you have available to you are the most basic, namely width and height.*

On the DVD that accompanies this book, you will find a project named “Chapter_12_SVG_in_Gambas3”. You can only open it in Gambas3 – if you want to look at the code in a text editor, you must set that folder’s View to “Show Hidden Files”, go into the “.src” folder, and open the text file “FMain.class”. This program not only explores how to handle SVG, but also demonstrates how to handle internal Gambas drawings and bitmap graphics in Gambas3. There are a couple of differences from the methods in Gambas2, although that may change in future Gambas3 releases. Several graphics are included, including one SVG image of a sucrose molecule that is a freely available public domain licensed image, downloaded from the Internet at:

<http://openclipart.org>.

Because Gambas3 may experience a few more changes, we will only look at the part of the included program that is the subroutine dealing with SVG graphics.

```
Public Sub DrawSVG_Click() 'Paint an SVG graphic to establish the correct syntax.  
Dim dadr As DrawingArea  
Dim dra As SvgImage  
dadr = New DrawingArea(FMain)  
dra = New SvgImage
```

15 See URL <http://www.w3.org/TR/SVG/intro.html> for more details on SVG.

16 See URL <http://www.inkscape.org/> for more details on Inkscape.

A Beginner's Guide to Gambas – Revised Edition

```
dadr.Cached = True 'The Cached property must be set to True. If  
'this is not done, the picture will not display.  
dadr.Border = 1 'The Border property must be set to 1, 2, 3, or  
'4. If Border is set to 0 (no border), the picture will not display.  
dadr.Painted = True 'The Painted property must be set to True. If  
'this is not done, the picture will not display.  
'now load it using the Load method  
dra = dra.Load("sucrose.svg")  
dadr.Move(205, 7, dra.Width, dra.Height)  
Paint.Begin(dadr) 'SVG graphics are drawn with Paint, not Draw.  
dra.Paint  
Paint.End      'As with Draw, Paint requires a Begin and an End method.  
End
```

Remember, this only works in Gambas3.

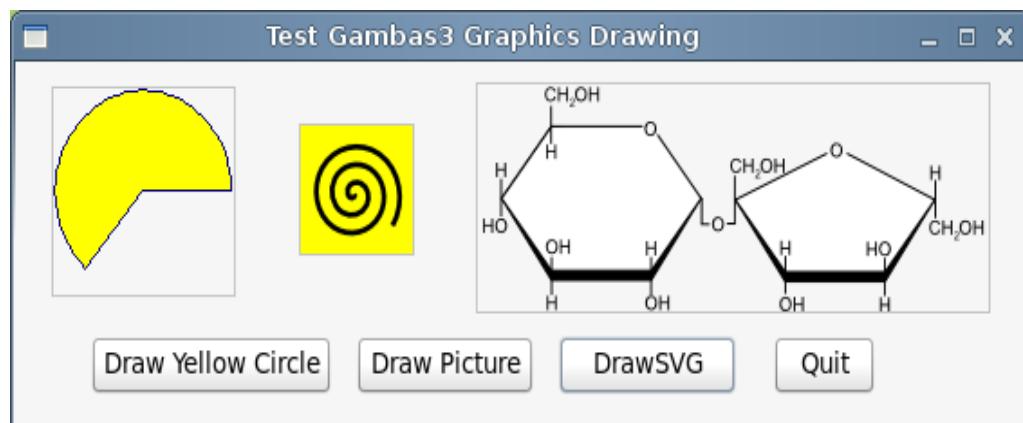


Figure 95: Loading and displaying graphics in Gambas3.

This figure is shown as implemented in the project “Chapter_12_SVG_in_Gambas3” on the included DVD. The picture of a sucrose molecule is an SVG graphic. We hope you have enjoyed our short little “tour de force” of Gambas drawing capabilities. This chapter has attempted to present you with most of the basics needed to use Gambas for almost any graphical application needs. In the next chapter, we will talk about error handling with Gambas.



Chapter 13 – Error Management

Error management is a key area for developers doing any kind of application development. Error management is a very powerful concept and can make development work much easier if it is used efficiently. As a Gambas developer, it is highly recommended that you adopt an appropriate strategy for error management that will enable you to build high quality and robust applications. Improper handling of errors can and will degrade the performance of your application, its perception by your user community, and ultimately, its overall acceptance. In order to understand error management, it is important to know the types of errors that can occur in a program. We will briefly explain the most common error categories in the following sections and try to point out some things you should watch for as you write your code.

General Concepts of Error Management

Errors can have many causes. Depending on the cause, the error might be recoverable. A recoverable error is one for which your application can identify the error cause and take some action to resolve or mitigate the problem. Some errors might be recoverable can be handled without the need to indicate to the user that an error was even encountered. For example, an error generated when a requested file does not exist is recoverable. The application can be programmed to display an error message in such an event. Errors such as validation errors, for which the application cannot continue processing but can provide error-specific feedback can also be considered recoverable if handled properly. For example, an error that occurs when a user enters text where a number is required can be considered recoverable because the application can recognize the error and redisplay the data field along with a message that provides information about the cause of the error, allowing the user to reenter data using correct input.

Error Handling

There are three key factors to consider when programming to avoid errors: prevention, detection, and recovery. Taking adequate steps to prevent errors will certainly help to ensure you have a happy user community. However, how do you prevent errors if they cannot be detected by the program before the user will suffer the consequences? Have you and your team taken the time to investigate and account for every possible keystroke, input, and data type that a user could enter? Really think about this while you are still in the design phase and again during the initial development of the test plans because it can come back to haunt you for many, many build and retest iterations. Finally, if the error does occur, what can be done to get the program back to the state of usability it was in **before** the error occurred?

Most errors are the result of inadequate design, poor programming practices, and poor testing. Poor programming practices include coding only the method of achieving the desired results without accounting for user error, mistakes in input, validation errors, etc. All programmers know how to develop data validation checks and usually try to put them off until

they get the code working. Making data validation a requirement bumps the priority of doing this work up a notch on the list of things the programmer has to do. Poor testing is epitomized by test teams failing to account for these types of factors in the design and construction of the product test plans and test cases.

Boundary-Related Errors

When a program executes, the program cursor moves from one part of the program to the next executing the coded instructions sequentially. The transition from one area to the next in the program is defined by the exit of one boundary condition and the entry to the next. This transitional state, or boundary, can include numeric, text, and operator boundaries. If the user is asked to enter a number from 0 to 99 in a program, what happens if the user enters the letter 'B' or types in 123 instead? What if the user enters the number 84 and the system tells you the number is too large, prompting the user to try entering a number from 0 - 99 once again? Does the program fail to validate input and attempt to process bad input? Have all possible boundary conditions been accounted for in design and test? Really?

Calculation Errors

There are an infinite number of conditions that can cause calculation errors. The programmer logic may have been in error or his or her formula for the calculation may be coded incorrectly. Rounding or truncation errors can significantly alter desired results. A good design process should incorporate pseudo-code for any and all calculated results to show how these formulas are intended to be implemented or used in production. Code reviews would check to see that they are indeed implemented as designed. Test cases would execute all boundary conditions against the code and verify the implementation is as rock-solid as it can be before turning it over to production for general use.

Initial and Later States

When a program executes subroutines and uses variables, these variables often must be initialized with default values. In Gambas, such defaults generally take the form of a zero value or a NULL value. If initialization is not done specifically by a developer, it is anyone's guess as to what the value of that variable will be when first called unless the variable is one of the data-types that Gambas initializes automatically. Remember, **these types of errors are a result of programmers not doing the job correctly**. As an example, let us assume that in the previous example above no variable initialization is what has happened when a routine calls for a variable to increment. The program processed 50 records, incrementing the value of the variable each time the routine is executed. When the value is output at the end of processing, it should be equal to 50. However, the un-initialized variable had a value of 122 (or some other random number) and the output shown to the user was 172. This describes what constitutes an initial state error.

To demonstrate later state errors, consider the same situation as above, but instead of passing the expected value of 50 to another module to use for processing, the value of 172 is

passed along. These errors are a bit harder to detect because the output from the called module is where the user thinks the error occurred. Programmers assigned to debug this problem have to trace through the execution path and follow the changing state of the variable until it is discovered to be a problem in the calling module, not the called one. The result of this is a great deal of time and effort went into discovering a problem that was avoidable. If the user reported this from a production system, project team credibility again takes a hit. As a program manager, you must ensure these types of situations are prevented by facilitating adequate, in-depth discussion in the design phase.

Control Flow Errors

Previously, we stated that when a program executes, the program cursor moves from one part of the program to the next, executing the coded instructions sequentially. Some execution instructions force the program to jump from one routine to another. This movement in execution is referred to as program or control flow. When a program does something wrong in control flow, it is generally a problem with the program logic or a programmer error in coding the design. Such logic errors are often uncovered during code reviews in the implementation phase. Sometimes, in testing they are discovered by building test cases to validate program flow based on the design specifications built in the design phase and the test plans built in the implementation phase. Any of these types of errors discovered by a user when a system is in production reflects a failure of the test team to adequately test the product. As a project leader, you are ultimately responsible for preventing this situation from occurring. TRY to work with the design team and test case developers as early in the process as possible to set proper expectations for the type of testing and the level of thoroughness you expect.

Errors in Handling or Interpreting Data

There are many ways data can become chock full of errors in a development process. However, almost all of these myriad ways data can go awry are attributed to human error. These types of errors can be a result of miscalculation, errors in logic, parameter passing errors, use of incorrect data types, etc. Almost anything that a programmer can do to make a mistake can cause an error in data. The best means of preventing such errors is a continual check of logic, review of expected and actual values, review of input and output data, etc. This is an ongoing process and it involves both user and developer.

From the user perspective, the data should be benchmarked against current means of producing the data. The user should continually review the results of output to ensure something has not run afoul and caused data to become bad. From the developer perspective, basic coding discipline should be enforced; range and boundary conditions need to be checked, parametric values need validation, data type validation must occur, and use of test data must be validated to be of the right format, current and in sync with production data, and usable for production output. It is always good for team leads to reinforce these concepts with developers and ensure periodic code reviews take place. It is essential to team success.

Race and Load Conditions

A race condition is best defined as one event preceding another. The trick is making sure the first condition should be first or the last condition is supposed to be last. Few programmers think about or even check for race conditions. These conditions are sometimes identified only after great manpower has been expended in doing traces into executable code, step by step, to see what is happening. When you hear the words irreproducible as testers are talking about this type of bug, interpret it to mean a possible race condition and ask someone if they have checked for it.

Load conditions are common and are the result of putting more work on the computer than it can bear to handle. A simple point of illustration is having a computer process millions of records. It is the perfect job for a computer but, when the time to process each record is more than a few milliseconds, the million records to be processed can take a great deal of time. For example, lets assume an ideal situation where your application does some complex validation, necessary to properly process a record, and that it takes 10 milliseconds to process each of these records in a perfectly managed test environment. This translates to a processing rate of 100 records a second. Not bad, but when you divide 100 into 1,000,000 records, you quickly find out that this job will take 10,000 seconds or 2.7 hours to run to completion. Again, acceptable in most cases, except that you must remember the machine is completely tied up with processing. No reports can be produced, no queries can be run, etc. In today's modern computing environment, this is not acceptable.

Lets take another view where real-world conditions are applied. In a test environment, where the 10 milliseconds are ideal computing conditions, we will compound the problem with user interaction and system overhead. Lets look at record retrieval times and factor that into the equation. Your 10 milliseconds are now 14.5 milliseconds per record. User queries and other user interactions interaction could easily add another 10 or so milliseconds per activity and now we are at 25 milliseconds of processing time for just a single record. We are now looking at 6.75 hours to run the same job that ran in 2.7 hours in the test lab. The point here is that testing should account for such load conditions before release into production. If a user knows before kicking off a production job that it will take 7 hours to run, they can go off and do other things or run the job at night. However, if the EXPECTATION is a 2.5 hour job, they will always be dissatisfied with performance because it is not what they expected.

Platform and Hardware Issues

Hardware issues are among the most frustrating a user can encounter. An example would be having a user select to print a report and watch it scroll by on the screen, at a million words a second. The printer device was not specified correctly or not specified at all. The user assumed a printer would print the report but device errors or lack of device handling caused the situation above. There are many types of devices which a programmer must account for and the testing process needs to account for user interaction on all of them. Aside from the condition cited above, programs can send the wrong codes to the right device, overrun the device with data, etc. Programmers and testers are responsible for ensuring these conditions

A Beginner's Guide to Gambas – Revised Edition

are accounted for and tested before your user community are exposed to them. All these types of errors can do is give you a bad mark in credibility as your users rate your performance.

Source, Version, and ID Control Errors

These types of errors are generally due to poor configuration management and an internal lack of enforcement of existing configuration management policies and procedures. An example of this is having a new programmer insert a beta version of code into a production stable release for a normal bug fix and then not telling anyone a change has been made in that module. Normal testing of the maintenance release will likely not catch the problem and it may go unnoticed for several subsequent releases.

As another example, this situation could be something as insidious as a data error that only shows up only when quarterly financials are produced. When there is no reason to be found as to why the company balance sheet is off by several hundred thousand dollars, much effort is expended in having senior developers perform code traces to find a problem. Someone will eventually discover that there is some beta code in a production module that has errantly rounded everything to the nearest dollar or some other similar stupid mistake. Good intentions aside, the best cure for these types of problems is good programmer discipline for configuration management and to have strong policy enforcement ingrained into an organizations corporate culture.

Testing Errors

Even testers make mistakes. They can interpret the logic of a test case improperly, they can misinterpret test results, fail to report glitches, etc. Testers may forget to execute test cases or report bugs that really don't exist. These folks are human too. That means their results are also subject to questioning. When your teams have worked hard to develop something and the test results come back skewed badly one way or another, question the test process.

Test Plan Reviews

For all but the most simple types of projects, a formal review process is generally required. This also requires participation from the users, the developers, the systems analysts, data analysts, and testers. It is best for all concerned if the formal test review include the most experienced users and developers present in an organization. Some users may not have the proper experience with a system to properly identify problems before they occur and this added experience can only benefit a review process. Ensure participation in a formal test plan review is based less on the availability of user and developer personnel and more on the contributions the best qualified developers and users can bring to the table. Now that we have a good understanding of the how and why of errors, lets talk about how we can manage such errors in Gambas.

Gambas Error management

Gambas provides built-in mechanisms to assist the programmer in error management. Such mechanisms are tools at a programmer's disposal that are often overlooked and, in many instances, not used at all. Gambas was designed with the concepts of simplicity and avoidance of errant logic in mind. Gambas programs you write should try to adhere to these concepts by using all of the means available to prevent errors. This means taking the time to write validation routines, working your code to track flow and provide output logs, etc. It is really not that much more work and will make a huge difference in the long run. Gambas can only provide an environment in which to develop, but, in the end, it is you, the coder, that is responsible for your product. Let's take some time now to learn how to use the Gambas features for error management.

TRY statement... IF ERROR

The TRY command provides a programmer with the means to perform an error check anywhere he or she suspects an error may occur. It is, in effect, a way to test the water before jumping in. You can try to execute the statement and, if an error occurs, it will be recorded by the Error class. This is done by setting property values that you can check. These property values include:

- ✓ Error.Class
- ✓ Error.Code
- ✓ Error.Text
- ✓ Error.Where

The Error.Class property will return to you the name of the class where the error occurred and the Error.Code property will tell you what the specific error code was. The text associated with the Error.Code is stored in the Error.Text property and the Error.Where property will tell you the offending line of source code (if possible). The error class also provides two methods, Error.Clear and Error.Raise that you can use to help track down problems. We will discuss the use of these two methods a little bit later in this chapter. The TRY statement is used to test for specific errors whereas the CATCH statement (described below) is more of a general purpose error catching mechanism. The general logic used for TRY is shown in the pseudo-code below:

```
TRY aStatement  
if ERROR THEN  
    Message.Error(error.text)  
endif  
'end of try
```

Let's create a quick example to demonstrate how to use try. Open the Gambas IDE and create a new graphical user interface project. Create a startup class form (FMain) and put an Exit

button on the form, named ExitBtn. Double-click on the button and add this code:

```
PUBLIC SUB ExitBtn_Click()
DIM a AS Integer
DIM b AS Integer

a = 2
b = 0

TRY PRINT a/b
IF ERROR THEN
    Message.Error("CLASS: " & Str(Error.class) & ", CODE: " & error.code & ", = " & error.text & " AT: " & error.where,
    "OK")
ENDIF
END
```

Save your work and execute the program. Click the Exit button and you should see something similar to this:



Figure 96: Error results caught with TRY.

Catch and Finally Statements

Catch is a general purpose error trap. It is executed when an error is raised anywhere between where a function starts to execute and where it ends. Catch provides a means to deal with errors that may occur in a given function and still allow the user to keep using the program. An error can be raised by the function itself, or by any other function it may call during execution. If the called function has a Catch block, the more nested the Catch block is, the more priority it has. In other words, if any error occurred in the nested section of code, the Catch in that section would be executed, not the Catch from the calling function. If an error is raised while the program is executing within the catch block itself, the error will normally be propagated outside the catch block because catch has no means of protecting itself. If there is a **Finally** part in the function, it must **precede** the catch part. Let's create a simple program to demonstrate this feature. Create a new project named *Chap13* as a GUI based project and when the IDE appears, create a startup class form named FMain. Add two buttons, Error and Quit, named ErrBtn and QuitBtn. Place the buttons on the bottom right corner of the form. Create a TLabel and name it TextLabel1. Put it at the top left corner of the form and stretch it all the way across and make it about one inch in height. Double-click

A Beginner's Guide to Gambas – Revised Edition

on the form and enter this code:

```
' Gambas class file

PUBLIC SUB _new()
TextLabel1.Text = "<center><strong>No errors present."
END

STATIC PUBLIC FUNCTION Run() AS Boolean
DIM hForm AS Form
hForm = NEW Form1
RETURN hForm.ShowModal()
END

PUBLIC SUB QuitBtn_Click()
ME.Close(TRUE)
END

PUBLIC SUB ErrBtn_Click()
DIM a AS Integer
DIM b AS Integer

a = 2
b = 0
PRINT a/b 'will generate a divide by zero error

FINALLY
TextLabel1.Text = "CLASS: " & Str(Error.class) & ", CODE: " & error.code & ", = " & error.text & " AT: " & error.where
TextLabel1.Refresh

CATCH
IF ERROR THEN
    Message.Error("CLASS: " & Str(Error.class) & ", CODE: " & error.code & ", = " & error.text & " AT: " & error.where,
"OK")
ENDIF
Message.Info("Cleared the Error", "OK")
TextLabel1.Text = "<center><strong>Error has been cleared."
END
```

When you run the program, you should see this as the opening screen:

A Beginner's Guide to Gambas – Revised Edition



Figure 97: CATCH test program opening screen.

Now, click the error button to generate our error and you should see this:

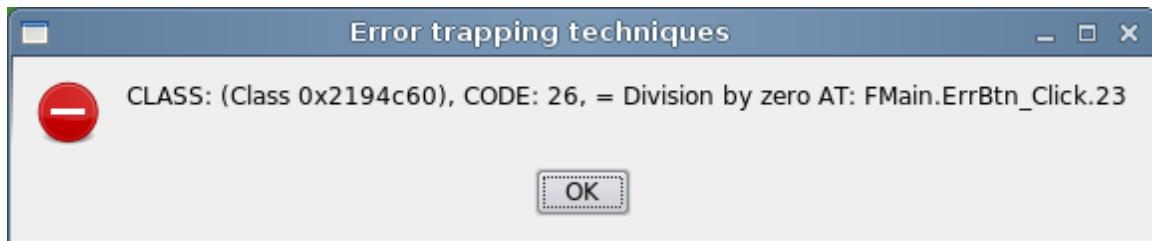


Figure 98: Div by Zero error caught by CATCH.

You should also notice that the main screen TextLabel has changed to this:



Figure 99: The TextLabel updated with error info.

Once you click the Ok button for the Error Message dialog, you will see this:



Figure 100: Info message that the error is cleared.

Now, click the OK button on the Info message and the TextArea is updated like this:



Figure 101: Main screen after error has been cleared.

This should give you a pretty good idea of how to use the Gambas error management features to help manage errors when they occur in your code (and they inevitably will occur). Not handling errors like the div by zero we used in our example will be a “show-stopper” in most circumstances. As a quick example, go back to our code, in the ErrBtn_Click subroutine and comment out these lines as shown below:

```
' FINALLY
' TextLabel1.Text = "CLASS: " & Str(Error.class) & ", CODE: " & error.code & ", = " & error.text & " AT: " & error.where
' TextLabel1.Refresh
'CATCH
' IF ERROR THEN
'   Message.Error("CLASS: " & Str(Error.class) & ", CODE: " & error.code & ", = " & error.text & " AT: " & error.where,
"OK")
' ENDIF
'Message.Info("Cleared the Error", "OK")
```

Now, run the program again and click the Error button. Here is what you get:



Figure 102: Default Gambas error dialog.

You can try to continue, but there is no guarantee that your program will work correctly or that it will even continue. This is definitely not what you would want your users to see. It just makes good sense to use the tools available in Gambas to prevent this when you can.

Now, lets move on to see how we can use events to help write better code.

Gambas Event management

Gambas is built around an event-driven programming paradigm. Traditional computer programming languages were designed to follow their own control flow changing flow direction (course) at branch points. In an event-driven paradigm, the control flow is largely driven by external events such as mouse movements, button clicks, etc. Generally, an event-driven system is preprogrammed using an event loop which systematically polls the system to look for information to process (*which could be any type of event, even the passage of time*). If an event is detected, it triggers a flag that can be checked in the event loop. This allows programmers to check for and respond to events by writing subroutines or function to process them.

How information on events is acquired by the system is not really important to the programmer. Common types of event input data can be polled in the event loop. Complex or proprietary events may require the use of interrupt handlers that react to hardware events, such as signal data traversing a Network Interface Card (NIC). Most event-driven systems use both techniques. Algorithms written by the programmer to respond to events ensure that when a given event is triggered, code to handle the event in a manner acceptable to the user is provided. This creates a layer of software abstraction that emulates an interrupt driven environment. Event-driven programs typically consist of a number of small programs called event handlers. Event handlers are called in response to events. A dispatcher is used to call the event handler. Most often, the dispatcher is implemented by using an event queue to hold unprocessed events.

Sometimes, event handlers can trigger events themselves. This can possibly lead to an event cascade (not good). Gambas, like most common graphical user interface programs, is programmed in an event-driven style. In Gambas, you have the capability of creating events for use in your programs. You can declare an event like this:

EVENT Identifier([Parameter #1 [, Parameter #2 ...]]) [AS Boolean]

This declaration will declare a *class event*. A class event is raised by a function call. Additionally, you can specify whether or not the event handler will return a (Boolean) value. If a return value is specified, a TRUE return result would indicate that the event was canceled.

In Gambas, the standard calling convention for an event handler would be *InstanceName_EventName* to invoke the method to be called by the event listener when such an event is raised. Let's add to our previous example to demonstrate the use of events. Add another button to the form for our previous example, named EventBtn. Double-click and add this code to the program:

EVENT MyEvent(Msg AS String) AS Boolean

```
PUBLIC SUB Event!Btn_Click()  
  
DIM bResult AS Boolean  
bResult = RAISE MyEvent("Raise an event")  
IF bResult THEN  
  
    TextLabel1.Text = "Cancelled the event"  
ELSE  
  
    TextLabel1.Text = "Raised the event"  
  
    bResult = NOT bResult  
  
    STOP EVENT  
ENDIF  
  
END
```

Save your work and run the program. Click the Event button and your display should look like this:



Figure 103: Our event was raised.

This chapter has explained the basic concepts of error handling in programming environments, Gambas in particular. The knowledge presented here will transcend the specific language you use to program and will be of great assistance in your programming endeavors over time. We hope this has helped you to understand how to better program your code and make your application more reliable and useful to the community of users you will support.



Chapter 14 – Mouse, Keyboard and Bit Operations

The mouse was designed to be a replacement for gathering keyboard input from users. While GUI design has advanced significantly since the early days of windowed interfaces, the evolution of the mouse has not. Probably the biggest change for the mouse was going optical and cordless. However, the basic functionality remains virtually the same. Provide an X,Y position, click a left, right, or middle button and detect it's state as up or down, and check the movement of the mouse wheel or change the delta value when the wheel moves. Gambas, of course, supports all of these basic features with an easy to code, logical approach. In the next few pages, we will show you how to master mouse and keyboard input. Finally, as a last exercise, we will create a program to show you how to use bit operators. Let's start with the mouse.

Mouse Operations

In Gambas, we can use the **Mouse** class to obtain information about a mouse event. This class defines the constants used with the mouse properties. The class is static and the properties supported are:

Alt	Button	Control	Delta
Left	Meta	Middle	Normal
Orientation	Right	ScreenX	ScreenY
Shift	StartX	StartY	X
Y			

Most of these properties are self-explained by their name, but a few need further explanation. The **Alt**, **Control**, **Meta** and **Shift** properties all return a TRUE value if those respective keys are pressed. **Normal** can be checked (tested) to see if any other special key was pressed. **Button** can be used to see if any button is pressed but you can use **Left**, **Right**, and **Middle** to determine if any specific mouse button has been pressed. All of these properties return TRUE if a button is pressed. **Delta** returns the (offset) value of the mouse wheel while **Orientation** can be used to check the direction (forward, backwards) of the wheel's turn. **ScreenX** and **ScreenY** return the absolute value of the cursor based on the desktop. StartX and StartY return the values of [Mouse.X](#) and [Mouse.Y](#) at the beginning of the event, while **X** and **Y** return the current position, relative to the window.

There is only one method for the **Mouse** class, **Move**. The **Move method** simply repositions the mouse cursor to the X,Y coordinates specified. Mouse events you can monitor in your program are **Mousemove**, **MouseDown**, **Mouseup**, and **Mousewheel**. Any change in the current state of those events allow you to create a coded response for that event. ***When coding mouse (or keyboard) operations, it is very important to always use the predefined***

constants. Using any other method of getting or setting properties is not guaranteed to be upward compatible from one Gambas version to the next. The Mouse class uses the following constants to specify the shape of the mouse cursor:

Arrow	Blank	Cross	Custom	Default
Horizontal	Pointing	SizeAll	SizeE	SizeH
SizeN	SizeNE	SizeNESW	SizeNW	SizeNWSE
SizeS	SizeSE	SizeSW	SizeV	SizeW
SplitH	SplitV	Text	Vertical	Wait

Let's create a short Gambas program to demonstrate how to use the mouse now. From the Gambas IDE, create a new project named MouseOps and make it a graphical user interface project, translatable with public controls. Create a new form, FMain, a startup class form. Next, add a drawing area named *da* to the top left corner of the FMain.form window. Resize it to be about one inch square. Now, let's add some code. First of all, when the program runs and the window opens, we want to set a caption that tells the user how to clear or exit the window. We also need to resize the drawing area to fit the form fully and set the cached buffer property to TRUE so our drawing can be seen by the user. Here is how to do this:

```
' Gambas class file
'when form opens at runtime, set a caption and resize the drawing area
PUBLIC SUB Form_Open()
    ME.Caption = "Press space to clear, mousewheel exits..."
    da.W = ME.W-5
    da.H = ME.H-5
    da.ForeColor = color.Black 'set the foreground color to black
    'set cached property to true so we can store the drawing buffer
    da.Cached = TRUE
    draw.Begin(da) 'start to draw
    'we will draw a horiz and vert line to section four quadrants
    draw.Line(1,da.h/2,da.W-1,da.h/2)
    draw.Line(da.W/2, 1, da.W/2, da.H - 1)
    draw.End
END
```

If the mouse moves, we want to change the cursor shape, depending on which quadrant of the drawing area the mouse is in. We will use the Mousemove event and code this:

```
'if the mouse moves, we will update the cursor shape depending on the
'quadrant the mouse is located at
PUBLIC SUB da_MouseMove()
    'check the top left quadrant
    IF mouse.X <= da.W/2 AND mouse.Y <= da.H/2 THEN
        da.Mouse = mouse.Pointing
```

A Beginner's Guide to Gambas – Revised Edition

```
'else we check the bottom left quadrant
ELSE IF
    mouse.X <= da.W/2 AND mouse.Y >= da.H/2 THEN
        da.Mouse = mouse.Cross
    'else we check the top right quadrant
ELSE IF
    mouse.X >= da.W/2 AND mouse.Y <= da.H/2 THEN
        da.Mouse = mouse.SizeNWSE
    'otherwise, it must be in the bottom right quadrant
ELSE IF
    mouse.X >= da.W/2 AND mouse.Y >= da.H/2 THEN
        da.Mouse = mouse.SizeNESW
    'if it is anywhere else, make it the default
ELSE
    da.Mouse = mouse.Default
ENDIF
```

Now that we have set the cursor shape, we will check to see if the user is pressing a mouse button to draw. If the left button is pressed, we will draw points. If it is the right button, we will draw tiny boxes.

```
'check mouse button is pressed to see which one is pressed and
'draw points for left btn, boxes for the right
IF Mouse.Left THEN
    'start to draw
    Draw.Begin(da)
    'little tiny dots
    Draw.Point(Mouse.X, Mouse.Y)
    Draw.End
ELSE 'if it is right button
    Draw.Begin(da)
    Draw.Rect(Mouse.X, Mouse.Y, 3, 3)
    Draw.End
ENDIF
END
```

Whenever the user moves the mouse wheel, we will terminate the application.

```
'if the user moves the mouse wheel, we will terminate the application
PUBLIC SUB da_MouseWheel()
    ME.Close
END
```

If the user has pressed and released a mouse button, we will reset the mouse cursor to the normal state.

```
'whenever a mouse button is released, reset mouse cursor to normal
PUBLIC SUB da_MouseUp()
    da.Mouse = mouse.Default
END
```

Finally, if the user presses any key, we will reset the drawing area by clearing it and redraw the quadrant by calling the same routine used to open the form:

```
'check kbd events to clear the drawing area and redraw the quadrants
PUBLIC SUB Form_KeyPress()
    da.Clear
    Form_Open()
END
```

That's it. When you run the program, you should see something like this:

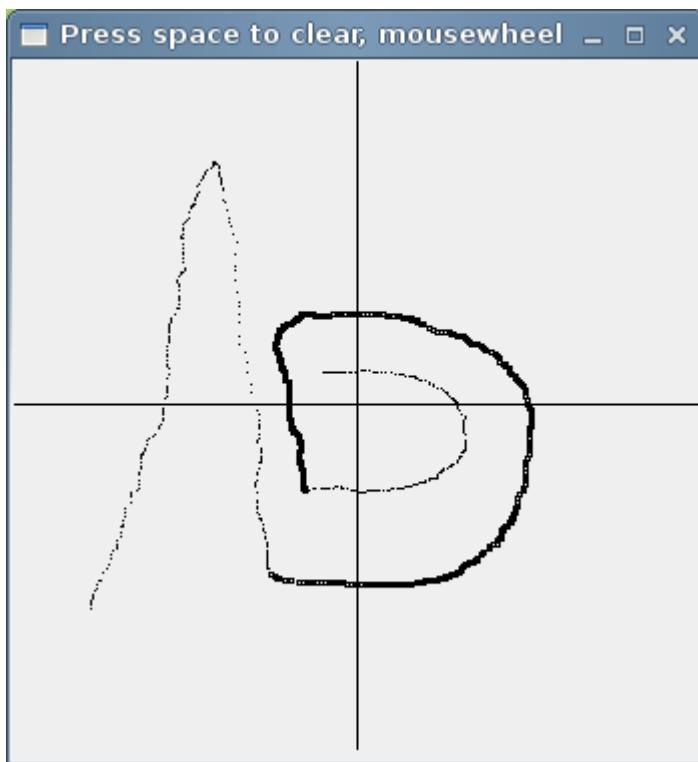


Figure 104: MouseOps program running.

As you move the mouse from quadrant to quadrant, notice how the cursor changes. Try both the left and right mouse buttons and press the spacebar to clear the drawing and restart. When you are satisfied everything works, move the mouse wheel and the program will

terminate. Now, let's learn about keyboard operations.

Keyboard Operations

This class is used for getting information about a keyboard event. It contains all of the predefined constants Gambas uses to represent the keys found on your keyboard. As we pointed out previously, you should never use the integer key values directly. You should always use the Gambas predefined constants. This class is static and it acts like a read-only array. You can declare an integer variable and obtain the value of a key constant using this calling convention:

```
DIM MyInteger AS Integer  
MyInteger = Key [ Key AS String ]
```

Properties that are supported for the Key class include **Alt**, **Code**, **Control**, **Meta**, **Normal**, **Shift**, **State** and **Text**. State can be used to test whether or not a special key is pressed and Text will return the single character representation for a given key. All the other properties work like those used with the Mouse class we described in the previous section. The Key class supports the following predefined constants:

For cursor keys: **Up, Down, Left, and Right**

For function keys: **F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12**

For shifted F-keys: **F13 F14 F15 F16 F17 F18 F19 F20 F21 F22 F23 F24**

For all other keys:

BackSpace	BackTab	CapsLock	Delete	End	Enter	Escape
Help	Home	Insert	Menu	NumLock	PageDown	PageUp
Pause	Print	Return	ScrollLock	Space	SysReq	Tab

Let's create another short Gambas program to demonstrate how to use the keyboard. From the Gambas IDE, create a new project named KbdOps and make it a graphical user interface project, translatable with public controls.

Create a new form, FMain, a startup class form. We will only need to use two labels in this program, a **TextLabel** and a **Label**. First, create a **TextLabel** named **TextLabel1** and make it about 1/2" high and stretch it across the width of the form. Set the **TextLabel1.Alignment** property to *center*.

Now, create a **Label** named **Label1** and place it below the **TextLabel1** control. Make it about two inches wide and 1/2" tall. Set the **Label.Text** property to "Press any key..." and set the control font size to 14 point and make it bold. Double-click on the form and the code window appears. Here is the code we will add to our program:

A Beginner's Guide to Gambas – Revised Edition

' Gambas class file

'we will check the key release event to see what to do

PUBLIC SUB Form_KeyRelease()

'whenever a key is released, check the code

SELECT key.Code

CASE key.Tab 'if it is a tab key

textlabel1.Text = "Key code is: " & key.Code & ", TAB."

CASE key.BackSpace 'or backspace

textlabel1.Text = "Key code is: " & key.Code & ", Backspace."

CASE key.CapsLock 'or caps lock

textlabel1.Text = "Key code is: " & key.Code & ", CapsLock."

CASE key.F1 'or a F-key

textlabel1.Text = "Key code is: " & key.Code & ", F1 key."

CASE key.F2

textlabel1.Text = "Key code is: " & key.Code & ", F2 key."

CASE key.F3

textlabel1.Text = "Key code is: " & key.Code & ", F3 key."

CASE key.F4

textlabel1.Text = "Key code is: " & key.Code & ", F4 key."

CASE key.F5

textlabel1.Text = "Key code is: " & key.Code & ", F5 key."

CASE key.F6

textlabel1.Text = "Key code is: " & key.Code & ", F6 key."

CASE key.F7

textlabel1.Text = "Key code is: " & key.Code & ", F7 key."

CASE key.F8

textlabel1.Text = "Key code is: " & key.Code & ", F8 key."

'if none of the above, is it printable? If so, show code and value

CASE ELSE

IF key.Code > 32 AND key.Code < 128 **THEN**

textlabel1.Text = "key code is: " & key.Code & ", " & Chr(key.Code)

ELSE 'otherwise just the code

textlabel1.Text = "key code is: " & key.Code

ENDIF

END SELECT

END

'check key down event to filter out shift, control, alt, etc.

PUBLIC SUB Form_KeyPress()

IF key.Shift **THEN**

textlabel1.Text = "Key code is: " & key.Code & ", SHIFT"

ELSE IF key.Alt **THEN**

```
textlabel1.Text = "Key code is: " & key.Code & ", ALT"
ELSE IF key.Control THEN
    textlabel1.Text = "Key code is: " & key.Code & ", CTRL"
ELSE IF key.Code > 32 AND key.Code < 128 THEN
    textlabel1.Text = "key code is: " & key.Code & ", " & Chr(key.Code)
ELSE IF key.Esc THEN 'if an ESC key is pressed
    textlabel1.Text = "key code is: " & key.Code & ", ESC"
ELSE
    textlabel1.Text = "key code is: " & key.Code
ENDIF
END

PUBLIC SUB Form_Open()
ME.Caption = " Keyboard Operations "
END
```



Figure 105: The Keyboard Operations program running.

As you can see, the code above is pretty straight-forward and simple. When you run the program, you should see something like this:

Bit Operations

Bits are the smallest elements used in computer operations. They are indivisible atoms of memory clustered in groups called bytes. A *byte* is composed of *n* number of bits, which is dependent upon the machine *word size*. For example, in an eight-bit system, one byte is composed of eight bits.

For modern computing systems used today, the operating system is built using 32 or 64 bit word sizes. The word size increases in order to accommodate a larger addressable memory space. In an older eight-bit system for example, the system can only address memory that can

A Beginner's Guide to Gambas – Revised Edition

be expressed in binary form as 11111111. In other words, the memory space is limited to a size that can be expressed with those eight bits. For an eight-bit systems memory ranges extend from the single-byte decimal representation of -65537 to +65538. In a modern 32-bit system, it takes the binary form of 111111111111111111111111111111 which can be expressed in decimal notation as -2,147,483,648 to +2,147,483,647. As you can see, the addressable range of memory with 32-bit systems is huge!

In applications level programming, there is generally not much occasion to operate at the bit-level when but occasionally the need does arise. Gambas provides all of the tools you need to test a bit, clear it, change (invert) it, or set it. Additionally, bitwise operations you can perform include rotating a bit right or left and shifting a bit right or left. The table below summarizes the bit operations you can perform in Gambas:

Gambas Bit Operations

Operator	Syntax	Action taken
BTst	Boolean = BTst(Number , Bit)	Test to see if a bit is set in <i>operand1</i> using <i>operand2</i>
BClr	Value = BClr(Number , Bit)	Clear a specific bit in <i>operand1</i> using <i>operand2</i>
BSet	Value = BSet(Number , Bit)	Return TRUE if the the bit specified in <i>operand2</i> of <i>operand1</i> is set.
BCchg	Value = BCchg(Number , Bit)	Returns <i>operand1</i> with the bit specified in <i>operand2</i> inverted.
Shl	Value = Shl(Number , Bit)	Returns <i>operand1</i> with the bit specified in <i>operand2</i> shifted left one bit position.
Shr	Value = Shr(Number , Bit)	Returns <i>operand1</i> with the bit specified in <i>operand2</i> shifted right one bit position.
Rol	Value = Rol(Number , Bit)	Returns <i>operand1</i> rotated left <i>operand2</i> bit positions.
Ror	Value = Ror(Number , Bit)	Returns <i>operand1</i> rotated right <i>operand2</i> bit positions.

Now, we will create yet another short Gambas program to demonstrate how to use the bit functions. From the Gambas IDE, create a new project named BitOps. Make it a graphical user interface project, translatable with public controls. Create a new form, FMain, your default startup class form.

Our program will take an integer number from 0 – 255 and display the hexadecimal and binary values for that number. It will allow you to select any one of the eight bits used to represent the integer number and perform whatever operation the button you click represents. The binary value of that integer will be displayed so you can compare that binary value to the value of the integer as originally specified.

For this program, we will need seven TextLabels, namedTextLabel1 through Textlabel7. In our example, for TextLabel5 and TextLabel7, we have set font style in the Properties tab to “Bold”. We will need two TextBoxes, TextBox1 and TextBox2. Finally, we will add nine buttons named **RolBtn**, **RorBtn**, **ShlBtn**, **ShrBtn**, **ClearBtn**, **SetBtn**,

TestBtn, **ChangeBtn**, and the **QuitBtn**. Layout the controls above on the form, as shown in the figure below:

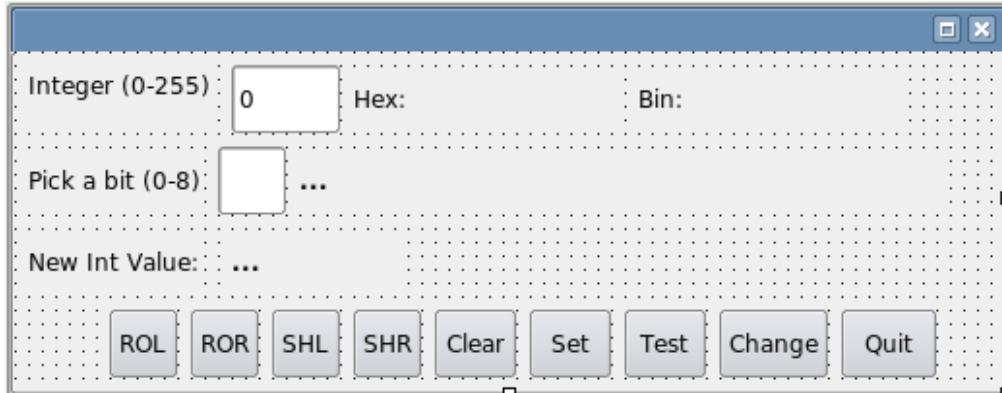


Figure 106: The BitOps program in design mode.

Once you have finished laying out the controls as shown in the figure above, double-click somewhere on the form and the code window will appear. Add the following code for the Form_Open subroutine:

```
' Gambas class file
PUBLIC SUB Form_Open()
    ME.Caption = "bitOpsns"
END
```

Now, the program will display the caption “bitOpsns” when it is executed. Let's code an exit procedure so the user can terminate the application next. Double-click on the Quit button and add this:

```
PUBLIC SUB QuitBtn_Click()
    ME.Close 'close the window
END
```

If the user enters any value in the TextBox2 field to change a bit, we need to update the form and reflect the changed values. We will worry about what to do with the new value a little later on in the code. For now, it is a simple call to the Refresh method we need:

```
PUBLIC SUB TextBox2_Change()
    FMain.Refresh 'refresh our display anytime the bit value changes
END
```

When the user specifies an integer value in the TextBox1 field, we will only react to the entry when the TAB key is pressed. That causes a LostFocus event, which is what we will code for. Here is what we need to do next:

```
PUBLIC SUB TextBox1_LostFocus()
'we need a local int var to work with
DIM myInt AS Integer
'convert our string text value to an int
myInt = CInt(TextBox1.text)
'check to see if it is in 8-bit range
IF Int(myInt) >= 0 AND (myInt < 256) THEN
    TextLabel2.Text = "Hex: " & Hex(myInt) 'convert hex value
    Textlabel3.Text = "Bin: " & Bin(myInt) 'convert bin value
    Textlabel7.Text = Str$(myInt) 'update our text fields
    FMain.Refresh 'refresh the form
ENDIF
END
```

When the user presses the clear button, we want to clear the bit specified for the integer represented by the first parameter.

```
PUBLIC SUB ClearBtn_Click()
DIM i AS Integer
DIM result AS Integer
DIM myInt AS Integer
'convert our string text value to an int
myInt = CInt(TextBox1.Text)
'if the integer is between 0 and 255 process, otherwise ignore
IF Int(myInt) >= 0 AND (myInt < 256) THEN
    'convert string value to an int
    result = CInt(TextBox2.Text)
    'clear the bit
    i = BCir(myInt, result)
    'update the label displays
    TexLlabel5.Text = "Bcir: " & Bin(i)
    TextLabel7.Text = Str$(i)
ENDIF
END
```

When the user presses the Set button, we want to set the bit specified for the integer represented by the first parameter. This is essentially the opposite of clear discussed above.

```
PUBLIC SUB SetBtn_Click()
DIM i AS Integer
DIM result AS Integer
DIM myInt AS Integer
```

```
'convert our string text value to an int
myInt = CInt(TextBox1.text)
'if the integer is between 0 and 255 process, otherwise ignore
IF Int(myInt) >= 0 AND (myInt < 256) THEN
    'convert string value to an int
    result = CInt(TextBox2.Text)
    'set the bit
    i = BSet(myInt, result)
    'update the labels
    TextLabel5.Text = "Bset: " & Bin(i)
    TextLabel7.Text = Str$(i)
ENDIF
END
```

The Bchg function is really just an invert operation. When our user clicks the Change button, we will execute this code:

```
PUBLIC SUB ChangeBtn_Click()
DIM i AS Integer
DIM result AS Integer
DIM myInt AS Integer

myInt = CInt(TextBox1.text)
IF Int(myInt) >= 0 AND (myInt < 256) THEN
    result = CInt(TextBox2.Text)
    i = BChg(myInt, result)
    TextLabel5.Text = "Bchg: " & Bin(i)
    TextLabel7.Text = Str$(i)
ENDIF
END
```

The Shl function will shift the value of MyInt n bits left where n is specified by the second parameter. When our user clicks the Shl button, we will execute this code:

```
PUBLIC SUB SHLBtn_Click()
DIM i AS Integer
DIM result AS Integer
DIM myInt AS Integer
myInt = CInt(TextBox1.text)
IF Int(myInt) >= 0 AND (myInt < 256) THEN
    result = CInt(TextBox2.Text)
    i = Shl(myInt, result)
    TextLabel5.Text = "SHL: " & Bin(i)
```

```
TextLabel7.Text = Str$(i)
ENDIF
END
```

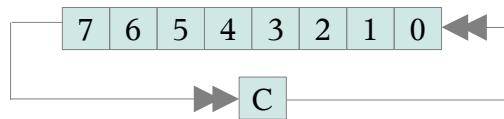
The Shr function is will shift the value of MyInt n bits right where n is specified by the second parameter. When our user clicks the Shr button, this subroutine will be executed:

```
PUBLIC SUB SHRBtn_Click()
DIM i AS Integer
DIM result AS Integer
DIM myInt AS Integer

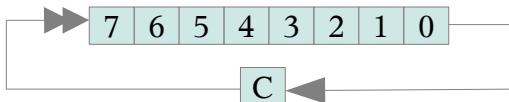
myInt = CInt(TextBox1.Text)
IF Int(myInt) >= 0 AND (myInt < 256) THEN
    result = CInt(TextBox2.Text)
    i = Shr(myInt, result)
    TextLabel5.Text = "SHR: " & Bin(i)
    TextLabel7.Text = Str$(i)
ENDIF
END
```

In older 8-bit systems, it was impossible to process numbers larger than 255 unless they were split up into 8-bit chunks. Operations were then performed on each piece of a number. After the operations were completed, it was necessary to reassemble each number back together to recreate the whole number. All of this was accomplished by using one tiny bit, the carry bit. The Gambas functions ROL (rotate left) and ROR (rotate right) are used to shift bits in binary numbers. These functions work conceptually like this:

The ROL Instruction



The ROR Instruction



ROL and ROR work such that the carry bit is shifted into the end bit that is left empty by the rotation process instead of inserting a zero value. ROL shifts the contents of a byte one place to the left. ROL does not place a zero into bit 0. Instead, it moves the carry bit from bit

A Beginner's Guide to Gambas – Revised Edition

position 7 into bit position 0 of the number being shifted. The 0 bit position was vacated by the shift rotation and the ROL instruction places bit 7 into the carry bit position. ROR works just like ROL, but in the opposite direction. It moves each bit of a byte right one position, placing the carry bit into bit 7 and bit 0 into the carry bit.

```
'rotate left
PUBLIC SUB ROLBtn_Click()
  DIM i AS Integer
  DIM result AS Integer
  DIM myInt AS Integer

  myInt = CInt(TextBox1.text)
  IF Int(myInt) >= 0 AND (myInt < 256) THEN
    result = CInt(TextBox2.Text)
    i = Rol(myInt, result)
    TextLabel5.Text = "ROL: " & Bin(i)
    TextLabel7.Text = Str$(i)
  ENDIF
END

'rotate right
PUBLIC SUB RORBtn_Click()
  DIM i AS Integer
  DIM result AS Integer
  DIM myInt AS Integer

  myInt = CInt(TextBox1.text)
  IF Int(myInt) >= 0 AND (myInt < 256) THEN
    result = CInt(TextBox2.Text)
    i = Ror(myInt, result)
    TextLabel5.Text = "ROR: " & Bin(i)
    TextLabel7.Text = Str$(i)
  ENDIF
END
```

If you want to check to see if a bit has been set, you can call Btst. This function will return TRUE if the bit specified in *operand2* of *operand1* is TRUE.

```
PUBLIC SUB TestBtn_Click()
  DIM i AS Boolean
  DIM result AS Integer
  DIM myInt AS Integer
```

```

myInt = CInt(TextBox1.text)
IF Int(myInt) >= 0 AND (myInt < 256) THEN
    result = CInt(TextBox2.Text)
    i = BTst(myInt, result)
    TextLabel5.Text = "Btst: " & i
ENDIF
END

```

That is all we have to do for our bit-fiddling program. Save your work and execute the program. Your results should be similar to this:



Figure 107: Our bitOps program running.

When you are satisfied that the program works as we planned, save your work. It would be a good exercise at this time to go back and modify the code to ensure only integer values are input into the two Textbox fields. Also, you could allow negative numbers to be entered to see what results would be produced. I will leave that exercise to those among you eager and ambitious enough to try – it is really not very difficult and we have already shown you how to do validation routines (remember, back in the collections chapter?). In the next chapter, we will cover database operations with Gambas. The ability to work with data in your programs is essential and we cannot overlook this vital aspect of programming in our journey through Gambas.



Chapter 15 – MySQL and Gambas

This chapter describes in detail the process of using the MySQL database from within the Gambas programming environment. For our purposes, we will assume that MySQL is already installed on your machine (or a machine accessible to you from your machine) and that a MySQL server is available for you to connect to from within your Gambas program. If this is not the case, then you will have to refer to the MySQL manual and figure out how to install the MySQL database distribution on your particular system. You should also be aware of the fact that Gambas will also work with PostgreSQL and SQLite. SQLite is not a true client/server architecture like MySQL and PostgreSQL. It is a flat-file database system and does not use the client/server database model. For our purposes in this chapter, we will stick to MySQL since it is probably the most widely used of the three database products.

The MySQL database package consists of the MySQL server and many client programs. The MySQL server is a program that stores and manages all of your databases within the MySQL environment. The client programs that are included with MySQL are too numerous to list here but it is important for you to understand that each client application serves a specific purpose related to the management and/or administration of data or metadata stored on the MySQL server. The client program we will be dealing with in this chapter is called `mysql` (*note it is spelled in lowercase letters*). This client program provides a command line interface (CLI) you can use to issue SQL statements interactively to the MySQL server and see the results displayed immediately. The main difference between **MySQL** and *mysql* is that **MySQL** is used to refer to the entire **MySQL** distribution package or to refer to the **MySQL** server while *mysql* refers to the client CLI program.

What purpose does it serve to have both a client and a server instead of a single program? The server and client programs are divided into different entities so you can use the client program(s) on your system to access data on a MySQL server that could be running on another computer. By separating the package into a server and *n* clients, it serves to separate data from the data retrieval interface. Before you begin working on this chapter, it is highly recommended that you familiarize yourself with MySQL by downloading the latest manual from the official MySQL web site¹⁷. At a minimum, complete the tutorial in Chapter 3 of the MySQL User Manual before proceeding with this chapter. Once you are familiar with how basic SQL operations work within the MySQL environment, you are ready to continue with this chapter, which focuses more on how to accomplish the same types of things the MySQL tutorial demonstrated, but from within a Gambas program.

In order to access databases within the Gambas environment, you need to ensure your project uses the `gb.db` component. When you create a new project, you need to go to the project properties dialog and you will find a tab for components. From within this tabbed dialog, you can select the components you need for your program. In the case of databases, you will need to check **gb.db** before you can use any of the classes defined by this component. As we stated previously, the `gb.db` component allows you to access the following database

17 See URL <http://www.mysql.com/>.

management systems: PostgreSQL, MySQL, SQLite2, SQLite3, Firebird, and ODBC. Because PostgreSQL and MySQL are client/server databases, it means your program will need to make a connection to a database server. SQLite is a flatfile-based database so there is no server process for your program to connect with. However, this also means that your program may need to use fully qualified pathnames for any database file it uses if the default path (*i.e.*, *the application path*) is not used.

The gb.db component has been designed to create an intermediate (abstracted) layer between the database server and your program. This allows you to **use the same code regardless of the database backend** you decide to use. There are a couple of caveats, of course. This abstracted approach only works if you have created your database by using the database manager or by using the gb.db component. Furthermore, you must use the Find, Create and Edit methods provided with the gb.db component. You cannot put SQL values directly in a request – you must use the substitution feature of the aforementioned methods to accomplish that. Finally, you cannot use the Exec method to send SQL requests directly to the server to access server-specific features. While this sounds fairly restrictive, in reality it is not a hindrance to your program. Given these few caveats, you can accomplish almost any database operation needed from within the Gambas environment. The classes that are supported in the gb.db component are *Blob*, *Connection*, *DB*, *Database*, *DatabaseUser*, *Field*, *Index*, *Result*, *ResultField*, *Table*. Note: Gambas 3 introduces a new class called *Connections*.

Blob Class

The Blob class is an object returned by a Blob field, which is a datatype used in SQLite3 to contain data of indeterminate format, like the Variant datatype in Gambas. This allows us to read or write data when the datatype that the variable will receive is not known. Blob has two properties: *Data*, which sets or returns the blob contents; and *Length*, which returns the length of the Blob contents in bytes.

Connection Class

The Connection class establishes your connection to the database. In order to successfully connect to a database, you must first create a connection object and set the necessary properties that will be used by the database. Next you must call the Open method. If you are connecting to a SQLite database and the Name property is null, a database is opened in memory. If the Name specified is an absolute path, the pathname is used. If the Name specified is a relative path and the Host property is null, the database is located in the application temporary directory */tmp/gambas.\$UID/sqlite*. Otherwise, Host will contain the database directory name and the database file path is the result of concatenating the Host and Name property values. This class is creatable and the calling convention is:

```
DIM hConnection AS Connection  
hConnection = NEW Connection()
```

Connection Properties

When the hConnection object is instantiated, it creates a new void connection object. You will then need to set properties for this object. The properties that you can use with a connection object include:

Charset

Charset is a read-only property that returns the charset used by the database. It is possible for you to store your strings in the database directly in UTF-8 format. However, you can also use this property combined with the Conv\$() subroutine to convert from UTF-8 format to the database charset format. Using the database charset comes at a cost in performance however. This is because you must use the Conv\$() subroutine each and every time you read or write string data to the database fields. For now, this is the only way you can do this if you want to use the string-based SQL functions and built-in sort routines of the underlying database management system (MySQL, PostgreSQL, or SQLite). Future versions of Gambas **may** automatically make the conversion between Gambas strings and the database charset.

Databases

The Databases property can be checked to obtain a collection object that will contain names of all databases that are managed by the database server. Syntax is:

PROPERTY Databases AS .ConnectionDatabases

The resulting collection of data (which is enumerable using the FOR EACH statement) would be the same as if you typed SHOW DATABASES from the mysql client CLI.

Error

This property returns the code of the last error raised by the database driver.

Host

You can use the Host property to set or get the host name of where the database server is located. The host name can be either a machine name or an IP address. The default host is named localhost. In SQLite, host name is not relevant since SQLite is not a client/server database system. However, from version 0.95 forward, it is possible for you to set the home path for the SQLite database by using this property. Syntax for using the Host property is:

PROPERTY Host AS String

Login

The Login property is used to set or get the user login data that is used for establishing

A Beginner's Guide to Gambas – Revised Edition

the connection to the database. Once again, this applies to MySQL and PostgreSQL, not SQLite. Since SQLite has no concept of users, access to a SQLite database is controlled by the file permission settings of the database file itself. This means that the Login property is always going to be set to the user id of the user that is executing the Gambas program that uses the SQLite database. Syntax is:

PROPERTY Login AS String

Name

The Name property sets or gets the name of the database you want to connect to. If you do not specify a name, a default system database is used. For PostgreSQL the default database is named *template1* and for MySQL it is named *mysql*. For SQLite, the default name is */tmp/sqlite.db*. For any database access, however, the user needs to have at least read and connect access privileges to use these databases. Sqlite will locate the database first by trying to use the full pathname if it has been given. If not, it will check to see if the Host variable has been set. If that is the case, SQLite will look at the path name that was set in the variable. If the environment variable **GAMBAS_SQLITE_DBHOME** has been set, then SQLite will use that as the current working directory. It is also possible to create and use a database in memory by setting the Name property to “**:memory:**” rather than “mysql” or “postgresql” or “sqlite”. Syntax is:

PROPERTY Name AS String

Opened

This Boolean property returns TRUE if the connection is opened. Syntax is:

PROPERTY READ Opened AS Boolean

Password

The Password property will return or set the password used for establishing a connection to the database. Calling convention is:

PROPERTY Password AS String

Port

The Port property is used to get or set the TCP/IP port used for establishing the connection to a database. The default port depends on the connection Type. Syntax is:

PROPERTY Port AS String

Tables

A Beginner's Guide to Gambas – Revised Edition

The Tables property is used to obtain a virtual collection that can be used for managing all of the tables of a database. Syntax is:

PROPERTY Tables AS .ConnectionTables

Type

The Type property represents the type of the database server you want to connect to. In Gambas, the currently supported database type are: "postgresql", "mysql", and "sqlite". The type property uses this syntax:

PROPERTY Type AS String

Users

This is a synonym for the Login property. Syntax is:

PROPERTY User AS String

The Users property returns a collection of all users registered in the database server. Like any other collection object, it can be enumerated with FOR EACH. Syntax is:

PROPERTY Users AS .ConnectionUsers

Version

The Version property is a read-only value that returns the version of the database that the driver has connected with. Syntax is:

PROPERTY READ Version AS Integer

The Concept of a Transaction

There are times when the order in which database actions (or queries) are executed is important. Sometimes, a program must ensure all queries in a group are run successfully and in order or process none of them at all. A classic example is taken from the banking environment. If a given amount of money (for example, 100 dollars) is taken from one account and posted to another account, we would expect the following actions to occur:

```
UPDATE account1 SET balance = balance - 100;
UPDATE account2 SET balance = balance + 100;
```

Both queries must execute successfully or neither must execute. Money cannot be transferred out of one account and fail to be posted to the other account. Both of these queries form a single transaction that is composed of two separate actions (*debit 100 from*

account1 and credit 100 to account2). A transaction is simply one or more individual database queries grouped together between **BEGIN** and **COMMIT** statements. Without a **COMMIT** statement, the transaction is not permanent and can be reversed with the **ROLLBACK** statement.

MySQL automatically commits statements that are not part of a transaction process. The results of any SQL UPDATE or INSERT statement that is not preceded with BEGIN will immediately be visible to all connections to the database because the commit is automatically performed. Most databases that are able to accomplish this are said to be **ACID-compliant**. The ACID model is one of the oldest and most important concepts of database theory. The four goals that must be met for any database to be considered reliable are **Atomicity**, **Consistency**, **Isolation** and **Durability** (ACID). Let's examine each of these four characteristics in detail.

Atomicity means database modifications must follow an “all or nothing” rule. Every transaction is considered an “atomic” unit. If any part of the transaction fails, the entire transaction fails. It is critical the DBMS maintain the atomic nature of transactions in any circumstance.

Consistency states only valid data will be written to a database. If a transaction is executed that violates database consistency rules, the entire transaction will be rolled back and the database will be restored to a state consistent with those rules. Conversely, if a transaction executes successfully, it will maintain a state of consistency.

Isolation requires multiple transactions that occur simultaneously not impact each other. For example, if User A issues a transaction against a database at the same time User B issues a transaction, both transactions should operate on the database in isolated bliss. The database should perform one transaction before the other. This prevents a transaction from reading intermediate data produced as a side effect of another transaction. Isolation does not ensure either transaction will execute first, just that they will not interfere with each other.

Durability ensures any transaction committed to the database will not be lost. This is accomplished via database backups and transaction logs that facilitate restoration of committed transactions despite any subsequent failures. Now, let's take a look at Gambas Connection class methods available to you in your programs.

Connection Class Methods

The connection class provides most of the methods you will need to establish a connection to a database and execute one or more transactions to find, add, change, or delete data.

Open/Close

The Open method is used to open a connection to the database. Prior to making the

call to Open, you should set the connection properties with the data necessary to use the database. Generally, at a minimum you will need to specify the type of database, the host, a login ID and password, and the name of the database you wish to use. The syntax for open is:

FUNCTION Open () AS Boolean

The code example below shows how to use the Open method to connect user *jrittinghouse* to a MySQL database named “*GambasBugs*” that is stored on Host *localhost*:

DIM \$hConn As NEW Connection

WITH \$hConn

**.Type = "mysql"
.Host = "localhost"
.Login = "jrittinghouse"
.Password = "ab32e44"
.Name = "GambasBugs"**

END WITH

TRY \$hConn.Open

IF Error THEN PRINT "Database cannot be opened. Error = "; Error.Text

To close the connection, simply invoke the Close method. In our example above, you could use this code:

\$hConn.Close

and the connection would be closed. In order to subsequently use any data stored in the database, you would need to re-open the database using the Open method.

Begin/Commit/Rollback

As we stated previously, a transaction is one or more individual database queries that are grouped together between BEGIN and COMMIT statements. Without a COMMIT statement, the transaction is not permanent and can be reversed with a ROLLBACK statement. Remember that MySQL automatically commits statements that are not part of a transaction process (defined by BEGIN and COMMIT). In Gambas, it is recommended that you use the Find, Create, and Edit methods to make changes to the database. This helps to maintain code independence and it allows you to write a single piece of code that will work with any database supported by Gambas. Gambas uses a **result object** to return the results of a SQL query.

Find

The Find method returns a read-only **Result object** used for querying records in the specified table. The calling convention for Find is:

```
FUNCTION Find (Table AS String [, Request AS String, Arguments AS , ... ]) AS Result
```

Request represents a SQL WHERE statement that is used for querying the table. *Arguments* are quoted as necessary (dictated by SQL syntax), and substituted inside the Request string. It works like the Subst() function in this case. Using the Find method allows you to write requests that are independent of the underlying database, meaning it will work with PostgreSQL or MySQL without any code change required.

Create

The Create method is used to build a read/write Result object which can be used to create records in a table. Standard calling convention is:

```
FUNCTION Create ( Table AS String ) AS Result
```

Edit

The Edit method returns a read/write Result object used for editing records in the specified table. *Request* is a SQL WHERE clause used for filtering the table, and the *Arguments* are quoted as needed by the SQL syntax and substituted in the Request string as we explained previously. Standard calling convention is:

```
FUNCTION Edit ( Table AS String [, Request AS String, Arguments AS , ... ]) AS Result
```

Here is a code sample to show how Edit() works:

```
DIM MyResult AS Result  
DIM sQuery AS String  
DIM iParm AS Integer  
  
sQuery = "Select * from tbIDEFAULT where id = "  
' we will insert a parameter value (12) at the end of the query string  
iParm = 12  
  
ShConn.Begin  
MyResult=ShConn.Edit("tbIDEFAULT", sQuery, iParm)  
MyResult!Name = "Mr Rittinghouse" ' Set a field value  
ShConn.Update ' Update with the new value  
ShConn.Commit ' make the changes permanent  
ShConn.Close ' close the database connection
```

Exec

The Exec method executes a SQL request and returns a read-only Result object containing the result of the request. Standard calling convention is:

FUNCTION Exec (Request AS String, Arguments AS , ...) AS Result

Request is a SQL WHERE clause and *Arguments* should be quoted as required by SQL syntax.

Limit

This method limits the number of records returned by the next query. After the query is executed, the number of returned records is reset to unlimited. This method returns the connection it applies to, so that you can write something like that:

```
DB.Limit(X).Exec(...)
```

Quote

The Quote method returns a quoted identifier you can use to insert in a SQL query. This identifier can be a table or field name. Syntax is:

FUNCTION Quote (Name AS String) AS String

The quoting mechanism is dependent upon the particular database server driver which makes this method necessary if you need to write database-independent code. Here is an example of how to use Quote:

```
' Returns number of records in a querys DBTable  
' is the name of the table. Since the name can include reserved  
'characters it needs to be surrounded by quote marks
```

```
rResult = $hConn.Exec("SELECT COUNT(*) AS iNumRecs FROM " & DB.Quote(sDBTable))
```

PRINT rResult!iNumRecs

Subst

The Subst method creates a SQL sentence by substituting its arguments in a format string. Syntax is:

FUNCTION Subst (Format AS String, Arguments AS , ...) AS String

Format is the SQL sentence, and **Arguments** are the arguments to substitute. The &1,

&2... patterns inside the Format string are replaced by respectively the SQL representation of the 1st, 2nd... Arguments. These arguments are quoted according to the underlying database SQL syntax. The arguments are used in the same manner that they are used in the Subst\$ function, discussed in Chapter 7.

For example:

```
PRINT DB.Subst("WHERE Name = &1 AND Date = &2", "Benoit", Now)
```

returns this output:

```
WHERE Name = 'Benoit' AND Date = '2011-02-15 11:51:33.043'
```

Result Objects

The result object is a class that is used to return the result of a SQL request. This class is not creatable and it acts just like a read-only array. It is enumerable using the FOR EACH statement. Declare and iterate a result object as follows:

```
DIM hResult AS Result  
FOR EACH hResult  
    'do something with the data  
NEXT
```

The code snippet below shows how you can get the value of a field in the current record of a Result object:

```
DIM MyResult AS Result  
DIM MyVariant AS Variant  
  
MyVariant = MyResult [Field AS String]
```

The ResultField class represents one of the fields of a Result object. You can use the Find method to select a particular field before using it. The properties this class supports are Length, Name, Result and Type. It is also possible to enumerate the ResultField data with the FOR EACH keyword. This class is not creatable. The code below shows you how to modify the value of a field in the current record of a Result object:

```
DIM MyResult AS Result  
DIM MyVariant AS Variant  
  
MyResult [Field AS String] = MyVariant
```

Properties that you can use with a result object include Available, Connection, Count,

Fields, Index, Length and Max. Available will return a TRUE value if the result points to an existing database record. Connection returns the parent connection object. Count tells you the number of records (rows) returned with the result object. Fields returns a collection of the fields in the database table returned with the result object. Index returns the index (current row pointer or cursor) of the current record, starting at zero. Length is the same as count and is used interchangeably. Max returns Result.Count-1.

Methods supported by the result object class include Delete, MoveFirst, MoveLast, MoveNext, MovePrevious, MoveTo, and Update. Most of these are self-explanatory, but the Update method is used to rewrite the current record to the database with any data changes that are specified. If you use MoveTo and specify an index that is void or invalid, it will return a TRUE value indicating an error. Now that we have a basic understanding of what a result object is and how to use it, let's see how Gambas allows you to find data in a database with the Find method.

DB Class

This class represents the current database connection. This class is static and all of its members are also static. Most of the properties of this class are the same as those of the Connection class so we will only cover those that are different, namely Current, Databases, and Debug. Current returns or sets the current connection. Databases returns a collection of all of the databases that are managed by the database server. Debug sets or returns TRUE if the database component is in debugging mode. When the debug flag is set, each query sent to any database driver is printed on the standard error output (generally, the console). The methods used in the DB class are the same as those used in the Connection class (Begin, Close, Commit, Create, Delete, Edit, Exec, Find, Limit, Open, Quote, Rollback and Subst) so we won't repeat that information here.

Database

The Database class is used to represent information pertaining to a database. This class is not creatable. The properties that you can use with this class include Connection, Name, and System. Connection can be used to obtain the parent connection object. Name will return the name of the database and System will return a TRUE value if the database is a system database. The database class only has one method, Delete, which will delete a database.

DatabaseUser

The DatabaseUser class is used to represent users of a database. This class is not creatable. The properties User class supports are Administrator, Connection, Name and Password. Administrator is a read-only property that returns TRUE if a user is a database administrator. Connection returns the parent connection object of the user. Name will return

the name of the user and Password will get or set the password associated with a user. When you read the property, you should get the password in an encrypted form. The DatabaseUser class has a single method, Delete which is used to delete a user from the database.

Field

The Field class is used to represent data for a table field. This class is not creatable. Properties that this class supports are Default, Length, Name, Table, and Type. Default returns the default value of a field. Length will return the maximum length of a text field. If the text field being checked has no limit specified, a zero value is returned. Name will return the field name and Table returns the table object in which the field was created. Type returns the datatype that the field represents. Gambas will return one of the following constants:

`gb.Boolean, gb.Integer, gb.Float, gb.Date or gb.String`

Index

The Index class represents a table index. This class is not creatable. The properties this class supports are Fields, Name, Primary, Table, and Unique. Fields returns a comma-separated list of strings representing the fields that make up the index. Name returns the index name while Primary will return a TRUE value if an index is the primary index of the table. Table will return the table object that owns this index. Unique returns TRUE if the index is unique.

Table

The Table class represents the definition of a database table. This class is not creatable. The properties that are supported by the Table class include Connection, Fields, Indexes, Name, PrimaryKey, System and Type. Connection can be used to obtain the parent connection object. Fields will return a collection of the fields of the table while Indexes will return a collection of the indexes of the table. Name will return the name of the table. PrimaryKey will return or set the primary key of the table. The primary key is a string array containing the name of each primary key field. System will return a TRUE value if the database is a system database. Type returns or sets the type of a table. This property is unique to MySQL and is only used with MySQL databases. This class only has one method, Update, which is called to actually create the table in the currently connected database.

The Database Example Program

The previous sections of this chapter have explained about the Gambas database component and the features provided therein. However, nothing substitutes working with real code in a real application. Gambas ships with an example database program written by Gambas founder Benoit Minisini, and this program is an excellent example of how to code

your program to use database within the Gambas environment.

We're going to work with a modified version of this program named "Database1", found on the DVD that accompanies this book. To begin, start Gambas and choose the Database1 program from the DVD. When the IDE appears, you will see that the program contains two forms, FMain and FRequest. FMain should look like this:

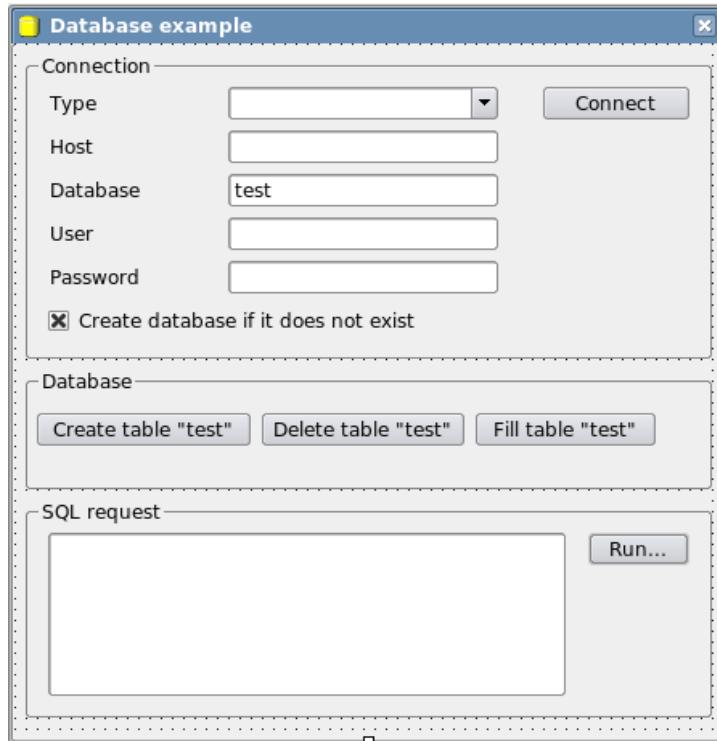


Figure 108: The FMain form in design mode.

This program will allow you to choose what database platform to use (i.e., PostgreSQL, MySQL, or SQLite) and specify the host name, database name, user ID and password. An option to automatically create a database exists and is activated whenever the checkbox is checked. Once you have successfully connected to the database, you have the option of creating a table, deleting it, or filling it with some test data. The SQL request box at the bottom of the form will allow you to perform SQL queries against the database. All in all, this program demonstrates almost everything you need to know to connect to any database using Gambas. The FRequest form is much simpler to build. It is shown in the figure below:

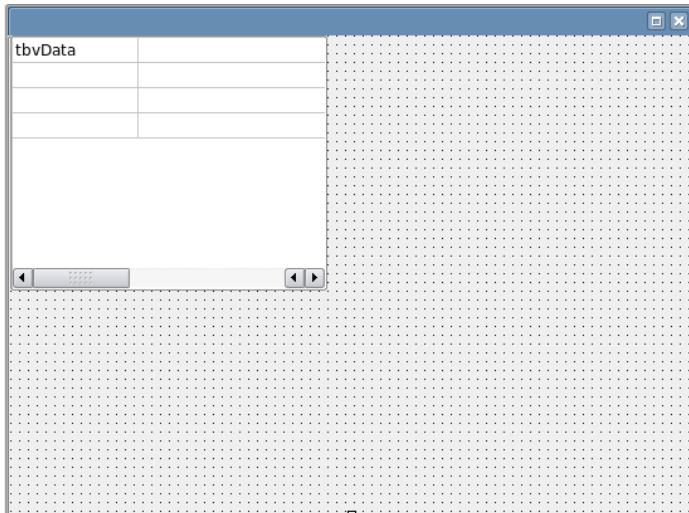


Figure 109: FRequest Form in design mode.

The FRequest form contains a single control, which we have not previously introduced, the TableView control. This control is found in the QT toolset we discussed previously. We will show you how to use this control when we discuss the FRequest form code below.

For now, let's jump into the code for the FMain form and see how things work.

' Gambas class file

\$hConn is declared as a private variable and will be used by all subroutines in this class file. Note that the variable is prefixed with a \$ symbol to indicate it is a private class variable. While not required, it is a good programming convention to follow.

PRIVATE \$hConn AS Connection

Whenever the user has filled out the data at the top of the form with their user name, host, password, etc., and clicks the connect button, the program will come to this subroutine and try to use that data to establish a connection to the database.

PUBLIC SUB btnConnect_Click()

```
'we need to have a string to work with so declare sName as local
DIM sName AS String
```

```
'now we get a little tricky. if a database was previously opened
'and is still open, this next line would close it. Using TRY
'would allow any error to be caught (see CATCH below).
'If there is not an open database to close, no harm was done
'and the code execution proceeds.
```

TRY \$hConn.Close

'we assign the data in the Textbox txtName to our string var sName
'and set the database connection properties to the values provided
'by the user by using the WITH/END WITH construct.

```
sName = txtName.Text
WITH $hConn
    .Type = cmbType.Text 'the type of database platform to use
    .Host = txtHost.Text 'host name of the database
    .Login = txtUser.Text 'USER ID
    Password = txtPassword.Text 'USER password
END WITH
```

'at this point, all the connection properties are set but
'we have not opened the connection yet. The following code
'segment will see if a database with the name specified by
'sName exists and, if not, it will create it if the checkbox to
'create database is checked.

```
IF chkCreate.Value THEN
    $hConn.Open 'open the connection
    'see if a database sName exists and if not, add it to the server
    IF NOT $hConn.Databases.Exist(sName) THEN
        $hConn.Databases.Add(sName)
    ENDIF
    'now close the connection
    $hConn.Close
ENDIF
```

'now, we are ready to open the database. Either it already existed
'and we open that database or we will open the new one we created
'if the checkbox was checked. First, set the Name property for the
'connection to sName and then call the Open method:

```
$hConn.Name = sName
$hConn.Open
'we set the enabled properties for both program forms to TRUE
'and place some default text in the SQL request box at the
'bottom of the FMain form:
```

```
frmDatabase.Enabled = TRUE
frmRequest.Enabled = TRUE
```

```
txtRequest.Text = "SHOW TABLES"
```

'if our TRY failed or if any error occurred during this subroutine,
'the CATCH statement would display an ERROR dialog with the text of
'the error message. DConv will convert the system charset to UTF-8.

CATCH

```
Message.Error(DConvError.Text))  
END 'of connectBtn_Click subroutine
```

If, after the database connection is established, the user would click the Create table “test” button, this subroutine would be executed.

```
PUBLIC SUB btnCreate_Click()
```

'declare a local Table variable

```
DIM hTable AS Table
```

'add the table to the database using the Add method

```
hTable = $hConn.Tables.Add(txtName.text)
```

'now add the fields of the table to the table we just added
'note that we specify the field name and data type for each
'of the fields we insert into the table. For string data,
'we also specify the maximum length of the string field.

```
hTable.Fields.Add("id", gb.Integer)  
hTable.Fields.Add("firstname", gb.String, 16)  
hTable.Fields.Add("name", gb.String, 32)  
hTable.Fields.Add("birth", gb.Date)  
hTable.Fields.Add("active", gb.Boolean)  
hTable.Fields.Add("salary", gb.Float)
```

'we can specify a primary key for the database table. In this
'case, the id field is used as we can create unique integer index
'data

```
hTable.PrimaryKey = ["id"]
```

'the call to the Update method commits the changes to the database
'and makes them permanent.

```
hTable.Update
```

A Beginner's Guide to Gambas – Revised Edition

```
'now we put up a simple dialog to inform the user we have  
'successfully completed the creation task.
```

```
Message.Info("Table " & txtName.Text & " has been created.")
```

```
'this code enables or disables the  
'buttons depending on the state of the database. For example,  
'if there is data already in the database table, it does not make  
'sense to fill that table again, so that button would be disabled.  
'however, the delete button would be enabled so the data could be  
'removed. If the database table is deleted, then it would be  
're-enabled and if the table is created and no data exists,  
'the fill button would again be enabled, etc. This prevents errors  
'from clicking on a button where an action could not be performed  
'successfully.
```

```
btnFill.Enabled = TRUE
```

```
btnDelete.Enabled = TRUE
```

```
btnCreate.Enabled = FALSE
```

```
'next, we add some default text to the SQL query box
```

```
txtRequest.Text = "Show TABLES"
```

```
'the CATCH statement would display an ERROR dialog with the text of  
'the error message if any error occurs in this subroutine.
```

```
CATCH
```

```
    Message.Error(DConv(Error.Text))
```

```
END 'of the create button subroutine
```

If, after the database connection is established, the user would click the Delete table “test” button, this subroutine would be executed.

```
PUBLIC SUB btnDelete_Click()
```

```
'remove the table
```

```
TRY ShConn.Tables.Remove(txtName.Text)
```

```
'put up a message to inform the user the table is gone
```

```
Message.Info("Table " & txtName.Text & " has been removed")
```

```
'enable or disable the buttons to make sense. If the table
```

A Beginner's Guide to Gambas – Revised Edition

```
'has been removed, it cannot be deleted again and it cannot  
'be filled since it does not exist. All that is left is to  
'recreate it and that button is enabled.
```

```
btnDelete.Enabled = FALSE  
btnFill.Enabled = FALSE  
btnCreate.Enabled = TRUE
```

```
'no table exists to query so blank out the SQL query text.  
txtRequest.Text = ""
```

```
'if an error occurs in the subroutine catch it here and show a dialog  
'message to the user.
```

```
CATCH
```

```
    Message.Error(Error.Text)
```

```
END 'the delete button subroutine
```

If the database connection has been established and the user has created the table, there needs to be some data added to make it useful. This subroutine is executed and will add semi-arbitrary data to the table. It will randomly select a first name from the array of five names provided and concatenate the counter number to the string “Name #” to serve as the last name. The birth date is created randomly by picking a value that adds a random number from 1-10,000 to the base date of January 1st, 1970. The record has an active flag, randomly set as well. Salary figures are randomly selected within the range of 1,000 to 10,000. Finally, the Update method will put all the data in the table and make the COMMIT to the database. Let's look at each line of code to see how easy this is accomplished in Gambas:

```
PUBLIC SUB btnFill_Click()
```

```
'we need an integer counter iInd to be our index and we  
'need a Result object to store our results
```

```
DIM iInd AS Integer
```

```
DIM rTest AS Result
```

```
'set the Busy flag to prevent interruptions to our process  
INC Application.Busy
```

```
'we are going to start the database transaction process with BEGIN
```

```
$hConn.Begin
```

```
'create the database table first
```

```
'rTest = $hConn.Create("test")
```

```
rTest = $hConn.Create(txtName.Text)
```

A Beginner's Guide to Gambas – Revised Edition

```
'now, set up a loop to create 100 records
FOR iInd = 1 TO 100
    'make the record id be the counter variable value
    rTest!id = iInd

    'randomly set the first name to be one of the five in the array
    rTest!firstname = ["Paul","Pierre","Jacques","Antoine","Mathieu"] [Int(Rnd(5))]

    'make the last name a catenated value with the integer index value
    rTest!name = "Name #" & iInd

    'randomly choose a date by adding a value from 1 - 10,000 to the
    'base date of Jan 1, 1970
    rTest!birth = CDate("01/01/1970") + Int(Rnd(10000))

    'set the active flag to either 0 or 1 (TRUE or FALSE)
    rTest!active = Int(Rnd(2))

    'randomly choose a salary figure from 1,000 to 10,000
    rTest!salary = Int(Rnd(1000, 10000))

    'update this record with the data semi-arbitrary data values
    rTest.Update
NEXT 'iteration of the loop

'commit all added records to the database
ShConn.Commit

'last thing to execute before leaving the subroutine
FINALLY
    'decrement the busy flag
    DEC Application.Busy

    'pop up a message to inform the user what we did
    Message.Info(txtName.Text & " has been filled.")

    'put a default SQL query in the SQL Query textbox
    txtRequest.Text = "select * from " & txtName.Text

    'fix our buttons to make sense
    btnFill.Enabled = FALSE
    btnDelete.Enabled = TRUE
    btnCreate.Enabled = FALSE
```

```
'if something goes wrong, abort all changes with the Rollback and  
'show the error message to the user  
CATCH  
    $hConn.Rollback  
    Message.Error(Error.Text)  
END 'of the fill data subroutine
```

If the user enters an arbitrary SQL request in the SQL query box at the bottom of the form, we will use the Exec method to perform the query and display the results of the request using the FRequest form. Here is how that is done:

```
PUBLIC SUB btnRun_Click()  
    'declare a result object to hold the results of the query  
    DIM rData AS Result  
  
    'declare a form variable so we can show the FRequest form  
    DIM hForm AS FRequest  
  
    'execute the query using the Exec method and assign the results to  
    'the result object rData  
    rData = $hConn.Exec(txtRequest.Text)  
  
    'pass the database connection handle and the result data to the  
    'FRequest form when it is instantiated  
    hForm = NEW FRequest($hConn, rData)  
  
    'now display the form to the user with the result data  
    hForm.Show  
  
    'come here if there is a problem in the subroutine and display a msg  
CATCH  
    Message.Error(Error.Text)  
END 'of sql query subroutine
```

Whenever our program runs and the FMain form opens, we need to instantiate our connection variable \$hConn. Whenever the form is closed, we will close the connection.

```
PUBLIC SUB Form_Open()  
    $hConn = NEW Connection  
END  
  
PUBLIC SUB Form_Close()
```

```
$hConn.Close  
END
```

Next, we need to take a look at the FRequest form and see how that code displays the result object data when it is invoked. Here is the code for that:

```
' Gambas class file  
' fRequest.class declares two private variables to use in this class  
' one for the connection, one for the result data.  
PRIVATE $hConn AS Connection  
PRIVATE $rData AS Result  
  
' a constructor routine will be used to receive the connection handle 'and the results of a query as parameters when  
FRequest is called by  
' the FMain form.  
  
PUBLIC SUB _new(hConn AS Connection, rData AS Result)  
' assign the hConn parameter to our private $hConn variable  
$hConn = hConn  
  
' assign the rData parameter to our $rData private variable  
' these assignments are made so the $ prefixed variables are  
' visible to the entire class, not just the constructor routine.  
$rData = rData  
  
' call our little subroutine to display the title in the form window  
RefreshTitle  
  
' the ReadData subroutine is used to populate our TableView control  
ReadData  
  
' resize the window to center on the desktop  
ME.Move(Int(Rnd/Desktop.W - ME.W)), Int(Rnd((Desktop.H - ME.H)))  
END 'of the constructor  
  
' this subroutine simply updates the window caption  
PRIVATE SUB RefreshTitle()  
  
' we need a local string variable  
DIM sTitle AS String  
  
' we will concatenate the connection name to the text for the caption  
sTitle = ("SQL Query Results ") & " - " & $hConn.Name
```

```
'and set the title property to be the string variable value  
ME.Title = sTitle  
END
```

The ReadData() subroutine is used to define the structure of the data passed in from the results object to the TableView control. It determines the number of fields from the results object and sets up the columns in the table, assigning the proper column (field) name and data type. The data type is determined by calling another subroutine, WidthFromType() to get that information.

The Qt library-based TableView control is obsolete in the Qt library, having been replaced with the QTable control. However, Gambas has added its own TableView control, so it is still usable and we will show you how it is done.

```
PRIVATE SUB ReadData()  
'this variable is declared but never used, you can comment it out  
DIM hTable AS Table  
DIM hField AS ResultField  
DIM iInd AS Integer  
  
'set the application busy flag to avoid interruptions  
INC Application.Busy  
  
'reset row count of the TableView control to zero  
tbvData.Rows.Count = 0  
  
'set the number of columns to be the same number  
'as the number of fields in the result object  
tbvData.Columns.Count = $rData.Fields.Count  
  
'now we will iterate through each field of the result object and  
'get the name of the field and the data type and set the TableView  
'column headings and field type/size as appropriate  
  
FOR EACH hField IN $rData.Fields  
WITH hField  
'this is a debug line that was commented out  
'PRINT .Name; ":"; .Type; ":"; .Length  
  
'this next line sets the column name to be the field name  
tbvData.Columns[iInd].Text = .Name
```

A Beginner's Guide to Gambas – Revised Edition

```
'here, the call to WidthFromType is used to determine what data
'type the field is and what the appropriate width should be
tbvData.Columns[iInd].Width = WidthFromType(tbvData, .Type, .Length, .Name)
END WITH
'increment our index counter
INC iInd
NEXT 'iteration of result data

'set the number of TableView rows to be the same as the number
'of rows of data in the result object
tbvData.Rows.Count = $rData.Count

'last thing we do in this subroutine is decrement the busy flag
FINALLY
DEC Application.Busy

'if there were any errors in this subroutine we would show a message
CATCH
  Message.Error("Cannot exec request." & "\n\n" & Error.Text)
END 'of ReadData subroutine
```

The next bit of code is activated whenever data is present. The *Data* event of the TableView control is used as the driver for activation of this subroutine. It inserts a row of data from the results object into the current row and column of the TableView control. Bear in mind that you don't ever call the data event directly. There is very little documentation available for the TableView control so nothing stated herein should be considered as the definitive answer for using it. However, the general approach to using the TableView control is to first load all data that is to be displayed into an array and prepare the TableView control with rows and columns as was done in the ReadData subroutine above.

The very process of filling the array and defining the columns in this manner forces the widget that is used in the TableView control to make an internal call to the *Data event* that actually does fill the cells with data from your array. ***It is up to you*** to code the assignment statement(s) that move data from the results object to the TableView row and column where you want the data displayed. The key thing to remember is that you ***must*** have an event handler, *tbvControl_Data(...)* somewhere in your class file to do this. In a TableView control, the data that is refreshed is only what is visible in the control. The control can display a huge amount of data without holding any of the data values directly in storage itself. That is what your program array is used for. While it sounds a bit awkward, it is a pragmatic solution that allows Gambas to display data from a database in a pretty efficient manner.

```
PUBLIC SUB tbvData_Data(Row AS Integer, Column AS Integer)
  'this gets to the correct row
  $rData.MoveTo(Row)
```

A Beginner's Guide to Gambas – Revised Edition

```
'this actually assigns data from the results object to the tableview  
tbvData.Data.Text = Str($rData[tbvData.Columns[Column].Text])  
END
```

If the form is resized by the user, we need to resize the control. This subroutine will do the trick.

```
PUBLIC SUB Form_Resize()  
    tbvData.Resize(ME.ClientW, ME.ClientH)  
END
```

When data is being read from the results object, this subroutine is called to determine what the data type is for each field of a row. The properties from the Field class are passed as parameters to this function and used in the select statement. This ensures the data placed in the individual cells of the TableView control is not truncated and will display properly, regardless of the font property setting established by the TableView control itself.

```
PRIVATE FUNCTION WidthFromType(hCtrl AS control, iType AS Integer, iLength AS Integer, sTitle AS String) AS Integer  
  
    DIM iWidth AS Integer  
  
    SELECT CASE iType  
        CASE gb.Boolean  
            iWidth = hCtrl.Font.Width(Str(FALSE)) + 32  
  
        CASE gb.Integer  
            iWidth = hCtrl.Font.Width("1234567890") + 16  
  
        CASE gb.Float  
            iWidth = hCtrl.Font.Width(CStr(Pi) & "E+999") + 16  
  
        CASE gb.Date  
            iWidth = hCtrl.Font.Width(Str(Now)) + 16  
  
        CASE gb.String  
            IF iLength = 0 THEN iLength = 255  
            iLength = Min(32, iLength)  
            iWidth = hCtrl.Font.Width("X") * iLength + 16  
    END SELECT  
    iWidth = Max(iWidth, hCtrl.Font.Width(sTitle) + 8)  
    RETURN iWidth  
END
```

A Beginner's Guide to Gambas – Revised Edition

Finally, if the user clicks on the “x” at the top right corner of the window to close the program, we come to this routine and close things down.

```
PUBLIC SUB Form_Close()  
    ME.Close  
END
```

When you execute the example Database program, it requires that you have a database system like PostgreSQL or MySQL installed on your system. You must have a user account and password to use the database systems. Given those caveats, run the program and log into the database. You can play around, create and fill the test table, and make queries with the program. At this point, you have learned all the basics for connecting to and using a database in Gambas. In the next chapter, we will cover the essentials of making your program available to users globally, a process called internationalization (I18N).



Chapter 16 – Global Gambas

Gambas has been designed from the very beginning to support an international community. Being an open source product, people from everywhere on the planet participate in the ongoing care and development of the Gambas product. The ability Gambas provides to developers to create a single code-base that will satisfy users from every country is a significant step in the recognition of a global community. Gambas allows developers to write programs that are easily translated to nearly any language, without having to rewrite code and without having to hire an expensive outsource firm to do the job. The processes of *Internationalization* and *Localization* are significantly enhanced by the Gambas approach. The following sections will provide an overview that every Gambas developer should be aware of in order to produce globally-acceptable code. We are, of course, all from one planet and should always consider the implications of what we place in an open-source community.

Internationalization

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes. Sometimes the term internationalization is abbreviated as i18n, because there are 18 letters between the first "i" and the last "n." An internationalized program, with the addition of localized data, can use the same executable worldwide. Textual elements, such as status messages and the GUI component labels, are not hard-coded in the program. Instead they are stored outside the source code and retrieved dynamically. Support for new languages does not require recompilation. Culturally-dependent data, such as dates and currencies, appear in formats that conform to the end user's region and language. It can be *localized* quickly.

Localization

Localization is the process of adapting software for a specific region or language by adding locale-specific components and translating text. The term localization is often abbreviated as l10n, because there are 10 letters between the "l" and the "n." Usually, the most time-consuming portion of the localization phase is the translation of text. Other types of data, such as sounds and images, may require localization if they are culturally sensitive. Localizers also verify that the formatting of dates, numbers, and currencies conforms to local requirements.

Universal Character Set (UCS)

ISO/IEC 10646 defines a very large character set called the Universal Character Set (UCS), which encompasses most of the world's writing systems. The same set of characters is also defined by the Unicode standard. To date, changes in Unicode and amendments and additions to ISO/IEC 10646 have been made in unison so that the character repertoires and

code point assignments have stayed synchronized. The relevant standardization committees have committed to maintain this very approach.

Both **UCS** and **Unicode** are code tables that assign an integer value to a character. There are several alternatives for how a string of such characters (or their integer values) can be represented as a byte sequence. The two most obvious encoding forms store Unicode text as sequences of either 2 or 4 bytes. These encoding schemes are designated as UCS-2 and UCS-4. In both of these schemes, the most significant byte comes first (Big Endian format). The Unicode characters are identified by number or “code point”, usually given in hexadecimal, so for example the Hebrew letter “he” is 5D4, usually written U+05D4.

Unicode currently defines just under 100,000 characters, but has space for 1,114,112 code points. The code points are organized into 17 “planes” of 2¹⁶ (65,536) characters, numbered 0 through 16. Plane 0 is called the “Basic Multilingual Plane” or BMP and contains symbols found to be useful. Generally speaking, it contains every character available to a programmer before Unicode came along.

The characters in the BMP are distributed in a “West to East” fashion, with the ASCII characters having their familiar ASCII values from 0 to 127, the ISO-Latin-1 characters retaining their values from 128 to 255. Then, the character sets move eastward across Europe (Greek, Cyrillic), on to the Middle East (Arabic, Hebrew), across India and on to Southeast Asia, ending up with the character sets from China, Japan, and Korea (CJK). Beyond this BMP exists planes 1 through 16. They are sometimes referred to as the “astral planes” because they are typically used for exotic, rare, and historically important characters.

ASCII files can be converted into a UCS-2 file by inserting ox00 in front of every ASCII byte. To make a UCS-4 formatted file, you must insert three ox00 bytes before every ASCII byte. Using either UCS-2 or UCS-4 on Unix platforms can lead to some very significant problems. Strings used with these encoding schemes can contain parts of many wide character bytes like “\0” or “/” that have special meaning in filenames and other C library functions. The vast majority of UNIX tools expect to operate with ASCII-based files and simply cannot read 16-bit words as characters. Simply put, UCS-2/UCS-4 are not suitable for Unix when used with filenames, text files, environment variables, code libraries, etc.

There are several other encoding forms common to both UCS and Unicode, namely UTF-8, UTF-16, and UTF-32. In each of these encoding formats, each character is represented as one or more *encoding units*. All standard UCS encoding forms except UTF-8 have an encoding unit larger than one octet.

For many applications and protocols, an assumption of an 8-bit or 7-bit character set makes the use of anything other than UTF-8 nearly impossible. Since UTF-8 has a one-octet encoding unit, it uses all bits of an octet but can still contain the full range of US-ASCII characters, all of which happen to be encoded in a single octet. Any octet with such a value can only represent a US-ASCII character.

UTF-8

UTF-8 was invented by Ken Thompson¹⁸. According to the story by Rob Pike who was with Ken at the time, it was developed during the evening hours of September 2nd in 1992 at a New Jersey diner. Ken designed it in the presence of Rob Pike on a placemat (see Rob Pike's UTF-8 history¹⁹). The new design replaced an earlier attempt to design a File System Safe UCS Transformation Format (FSS/UTF) that was circulated in an X/Open working document in August of the same year by Gary Miller (IBM), Greger Leijonhufvud and John Entenmann (SMI) as a replacement for the division-heavy UTF-1 encoding presented in ISO 10646-1²⁰. Within a week's time in September 1992, Pike and Thompson made AT&T Bell Lab's Plan 9 the first operating system in the world to use UTF-8 encoding. They reported this at the USENIX Winter 1993 Technical Conference²¹. FSS/UTF was briefly also referred to as UTF-2 and later renamed UTF-8 and pushed through the standards process by the X/Open Joint Internationalization Group XoJIG.

UTF-8 encodes UCS characters as a varying number of octets. The number of octets and the value of each depend on the integer value assigned to the character in ISO/IEC 10646 (*the character number, a.k.a., its code position, code point or Unicode scalar value*). This encoding form has the following characteristics²² (values in hexadecimal):

- Character numbers from U+0000 to U+007F (US-ASCII set) correspond to octets 00 to 7F (7 bit US-ASCII values). A direct consequence is that a plain ASCII string is also a valid UTF-8 string.
- US-ASCII octet values do not appear otherwise in a UTF-8 encoded character stream. This provides compatibility with file systems or other software (e.g., the printf() function in C libraries) that parse based on US-ASCII values but are transparent to other values.
- Round-trip conversion is easy between UTF-8 and other encoding forms.
- The first octet of a multi-octet sequence indicates the number of octets in the sequence.
- The octet values C0, C1, F5 to FF never appear.
- Character boundaries are easily found from anywhere in an octet stream.
- The byte-value lexicographic sorting order of UTF-8 strings is the same as if ordered by character numbers. Of course this is of limited interest since a sort order based on character numbers is almost never culturally valid.
- The Boyer-Moore fast search algorithm can be used with UTF-8 data.
- UTF-8 strings can be fairly reliably recognized as such by a simple algorithm, i.e., the probability that a string of characters in any other encoding appears as valid UTF-8 is low, diminishing with increasing string length.

18 <http://www.cs.bell-labs.com/who/ken/>

19 <http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt>

20 <http://64.233.167.104/search?q=cache:U4zV7dcaq0EJ:www.faqs.org/ftp/rfc/pdf/rfc2279.txt.pdf+Greger+Leijonhufvud&hl=en>

21 USENIX Technical Conference held in San Diego, CA, January 25-29, 1993, published in the Conference Proceedings, pp. 43-50.

22 See URL <http://www.faqs.org/rfcs/rfc2279.html>.

The **UTF-8** encoding is formally defined in ISO 10646-1:2000 Annex D. It is also described in RFC 3629 as well as section 3.9 of the Unicode 4.0 standard. In order to use Unicode under Unix-style operating systems, UTF-8 is the best encoding scheme to use. This is because UCS characters U+0000 to U+007F (ASCII) are encoded as bytes ox00 to ox7F (for ASCII compatibility).

This means files and strings that contain only 7-bit ASCII characters will have the same encoding under both ASCII and UTF-8. All UCS characters that fall in the range greater than U+007F are encoded as a multi-byte sequence, each of which has the most significant bit set. Therefore, no ASCII byte (ox00-ox7F) can appear as part of any other character.

The first byte of a multi-byte sequence that represents a non-ASCII character will always fall in the range of oxCo to oxFD and it indicates how many bytes will follow for this character. Any further bytes in a multi-byte sequence will be in the range from ox80 to oxBF. This allows easy resynchronization and makes encoding stateless and robust against missing bytes. All possible 231 UCS codes can be encoded.

UTF-8 encoded characters may theoretically be up to six bytes long. The sorting order of Big Endian UCS-4 byte strings is preserved. The bytes oxFE and oxFF are never used in the UTF-8 encoding. The following byte sequences are used to represent a character. The sequence to be used depends on the Unicode number of the character:

U-00000000 – U-0000007F: 0xxxxxx

U-00000080 – U-000007FF: 110xxxxx 10xxxxxx

U-00000800 – U-0000FFFF: 1110xxxx 10xxxxxx 10xxxxxx

U-00010000 – U-001FFFFF: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

U-00200000 – U-03FFFFFF: 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

U-04000000 – U-7FFFFFFF: 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

The xxx bit positions are filled with the bits of the character code number in binary representation. The rightmost x bit is the least-significant bit. Only the shortest possible multi-byte sequence which can represent the code number of the character can be used.

Note that in multi-byte sequences, the number of leading 1 bits in the first byte is identical to the number of bytes in the entire sequence.

How to translate in Gambas

To translate Gambas into your own language, open the Gambas project in the IDE, and click on Translate... in the Project menu, as shown in the figure below:



Figure 110: Choosing the Translate option in Gambas.

Next, you must select the target translation language in the combo-box of the translate dialog box, as shown below. Click on “New” and choose Esperanto (Anywhere!):

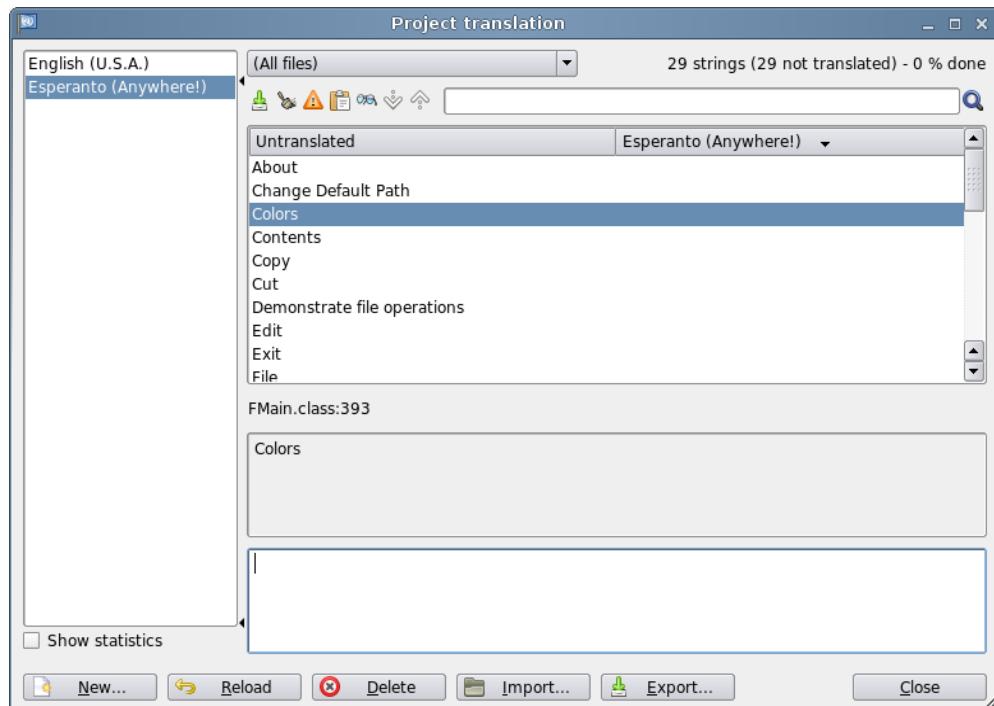


Figure 111: The Translate Dialog in Gambas.

Select a string from the list of untranslated strings and enter its translation in the text field shown at the bottom of the dialog. You need to repeat this process until every string in your application has been translated. When you have finished, click on the Close button. It is not necessary to complete all of the strings at one sitting. You can translate the project in

several attempts. The toolbar icons you see at the top of the Translate dialog allow you to:

- ✓ Save the current translation.
- ✓ Reload the current translation (Note: any changes are lost when you reload).
- ✓ Delete the current translation.
- ✓ Duplicate an existing translation.
- ✓ Export the current translation to a *.po* file.
- ✓ Merge a translation file.
- ✓ Verify a translation by checking to see if every symbol character is preserved.

There are certain occasions where you may not want to ever translate a string. In Gambas, you can indicate a string should not be translated by entering a minus sign character as the translated string. Your translation is stored in a **.po* file in the *.lang* directory of the project. The name of the *.po* file depends on the language translated. For example, the French translation file is named *fr.po*.

Whenever the latest Gambas version changes, any un-translated strings in your program could change. However, any strings you have translated will be preserved. If an un-translated string disappears completely, the translation will also disappear and if a new un-translated string is added, it gets an empty or null translation string assigned to it. All other un-translated strings will keep their corresponding translation strings. While the string translation feature of Gambas allows you to easily maintain string conversions, other locale specific things such as dates, number formats, etc. are left to you to manage. It is hoped that Gambas2 will address these features when it is released. Until that time arrives, here is a short, generalized checklist for helping you localize your application:

- ✓ Ensure all messages, icons, and human readable content are stored in external resource files and can be easily translated.
- ✓ The translated message or resource files should be dynamically loaded by the application, depending on the current language and locale settings for the session.
- ✓ Date/time, sort order, numeric and monetary formatting etc. are in the target language. The sort order should be configurable, depending on the language of the user.
- ✓ Target language characters can be correctly entered and displayed by the user and they can be read from and written to the native file system of the target platform.

According to Huang et. al.,²³ you should consider creating an I18N testing checklist for your application. Some of the questions they recommend you consider include the following:

- ✓ Does the program design account for international considerations?
- ✓ Can users enter text, accelerators/shortcut-key combinations on an international keyboard?
- ✓ Can a user enter dates, times, currencies, and numbers using international formats?

²³ Huang, E., Haft, R. and Hsu, J. “Developing a Roadmap for Software Internationalization”, Whitepaper dated October 2000, retrieved from URL [www.symbio-group.com/doc/Developing a Roadmap for Software Internationalization.pdf](http://www.symbio-group.com/doc/Developing%20a%20Roadmap%20for%20Software%20Internationalization.pdf) on 5 Oct 05.

A Beginner's Guide to Gambas – Revised Edition

- ✓ Can a user successfully cut/paste text with accented (or double-byte) characters?
- ✓ Can the application work correctly on different types of hardware sold in the target market?
- ✓ Does the application work correctly on localized operations systems?
- ✓ Can users type European accented (or double-byte) characters in documents/dialog boxes?
- ✓ Can documents created in a target language be opened in other languages and vice-versa?
- ✓ Can users save/print files with accented (or double-byte) characters?

While there is no single checklist that can account for the wide degree of variation from application to application, consideration of these basic issues will likely go far in helping you to reach a global audience. The Internet has made the world a much smaller place and the audience for applications have become much larger. You cannot simply assume everyone using your application is English-speaking and uses the same keyboard or will understand your icons. Different pictograms mean different things depending on the locale in which your user resides. Most important of all, make the effort to accommodate the broadest audience possible and your users will most likely allow for a few mistakes here and there. Not making such an effort will generally leave them with the impression that you could care less and they will probably feel the same way about using your program.



Alphabetical Index

Add method.....	87	Default button.....	59
AND operator.....	36	Default property.....	64
arithmetic operators.....	30	Delete method.....	68
assignment statement.....	30	Design property.....	64
Background property.....	61, 66	Desktop.Charset.....	141
Boolean data-types.....	25	Dialog class.....	101
Border property.....	62	Dialog Title property.....	102
buil.....	29	division by zero.....	34p.
Byte data-type.....	26	Division by Zero.....	36
Cancel button.....	59	DO [WHILE] LOOP.....	50
Cancel property.....	62	dot notation.....	30
Caption property.....	62, 79	double-click event.....	76
CASE ELSE block.....	49	Drag method.....	68
CASE statement.....	48, 102, 113	Drop property.....	64
CATCH statement.....	305	Enabled property.....	65
CheckBox class.....	91	END keyword.....	33
CheckBox control.....	90p.	END WITH instruction.....	30
Checked property.....	98	Enter event.....	72
Chr(9).....	135p.	Error management.....	300
class event.....	310	Error.Class property.....	305
class hierarchy.....	22	Error.Clear.....	305
Click event.....	74	Error.Code property	305
Collections.....	55	Error.Raise.....	305
ComboBox arrays.....	87	Error.Text property.....	305
ComboBox control.....	84p.	Error.Where property.....	305
ComboBox items.....	87	event driven language.....	45
comparison operators.....	30	event handler.....	76
conditionals.....	45	event-driven programming.....	310
Conditionals.....	45	event-driven system.....	310
Constant.....	28	Expand property.....	65
Container class.....	60	Finally.....	306
Control class.....	59	float data-type.....	26
Control Groups.....	97	Float data-type.....	26
controls.....	57	focus control.....	69
copyright notice.....	2	Font class.....	107pp.
Copyright Notice.....	1p.	Font property.....	65
Cursor property.....	63	FOR EACH.....	50
date data-type.....	27	FOR EACH construct	55
Date data-type.....	27	FOR statement.....	49p., 55
Dblclick event.....	76	Foreground property.....	65
DEC operator.....	38	Frame class.....	89

A Beginner's Guide to Gambas – Revised Edition

Gambas component library.....	88	Message class.....	111
Gambas constants.....	28	MessageBox control.....	111
Gambas Constants.....	29	MIME TYPE.....	68
Gambas data-types.....	24, 28	MOD function.....	35
Gambas Development Environment.....	64	MOD operator.....	36
Gambas IDE.....	57	Mouse property.....	66
Gambas Interpreter.....	14	Move method.....	69
GAMBAS interpreter.....	141	Native array.....	53
Gambas keywords.....	45	NEW keyword.....	25
Gambas ToolBox.....	57	New Project Wizard.....	31, 70
Gambas Wiki.....	14	Next property.....	66
GOTO instruction.....	49	NEXT statement.....	50
Grab method.....	68	NEXT Statement.....	49
Graphical User Interface classes.....	57	NOT operator.....	42
GTK+.....	15	object data-type.....	27
GTK+5.....	14	Object data-type.....	27
H property.....	66	OpenContent License.....	1p.
Handle property.....	66	OpenFile Dialog.....	102, 116
handles.....	60	OR operator.....	37
Height property	66	Panel class.....	92
Hide method.....	68	Panel control.....	92
HSV function.....	61	Parent property.....	67
html.....	68	Picture class.....	67
HTML.....	19, 78pp.	Picture control.....	76
Hungarian Notation.....	26	picture object.....	77
IF statement.....	46, 98	Picture object.....	64
IF Statement.....	46	Picture property.....	67
IF THEN ELSE.....	109	PNG file.....	77
INC operator.....	38	Pointer datatype.....	28
infinite loop.....	51	Power operator.....	36
Integer data-type.....	26	predefined constants.....	61, 65, 102, 148
Java.....	53	Previous property.....	67
Java-like arrays.....	53	PRINT statement.....	34, 45, 50
LABEL.....	49	PRINT Statement.....	45
Leave event.....	72	PRIVATE keyword.....	25
Left property.....	66	ProgressBar control.....	71
LIKE.....	39, 43	Project Explorer.....	18pp., 31
Linux.....	141	PUBLIC keyword.....	25, 28
ListBox control.....	87p.	qt.....	57
Load conditions.....	303	Qt.....	14p., 57, 141
localization.....	138pp., 142, 145	Qt6.....	14
looping structures.....	49	QUIT command.....	33
Lower method.....	68	race condition.....	303
Menu Editor.....	96p.	RadioButton class.....	93
MenuItems.....	96p.	RadioButton control.....	92p.

A Beginner's Guide to Gambas – Revised Edition

Raise method.....	68	Textbox control.....	82
recoverable error.....	300	TextBox control.....	79, 82
Refresh method.....	69	ToggleButton control.....	90
REPEAT and UNTIL.....	52	ToolBar.....	22
Resize method.....	69	ToolBox.....	19
RETURN command.....	33	ToolBox pane.....	22
RGB function.....	61	ToolTip.....	22
SaveFile Dialog.....	102, 117	ToolTip property.....	67
ScreenX property.....	67	Top property.....	67
ScreenY property.....	67	Treeview.....	103
SELECT statement.....	48, 104, 113	TreeView.....	18, 31, 62
SelectColor Dialog.....	102, 104	TRY.....	305
SelectDirectory Dialog.....	102, 118	TRY statement.....	305
SelectFont Dialog.....	107pp.	UTF-8 charset.....	135, 141
SetFocus method.....	69	Value property.....	67
SetFocus() method.....	67	variant data-type.....	27
Short data-type.....	26	Variant data-type.....	27, 55
Show method.....	68	VB.....	26, 33
STATIC keyword.....	25	Visible property.....	67
Status Bar.....	19	W property.....	66
STEP keyword.....	49	WAIT.....	69
String constants.....	29	Warning messages.....	114
string data-type.....	29, 55, 124	WHILE ... WEND.....	52
String data-type.....	27	Width property.....	66
string functions.....	124	Window property.....	68
string operators.....	30	WITH.....	30
subMenus.....	97	WITH keyword.....	30
System class.....	88p.	X property.....	66
t-in constants.....	29	X-Windows.....	77
Tag property.....	67	XOR operator.....	38
terminal application.....	31, 124	Y property.....	66
Text property.....	62, 67		

Appendix A

Gambas online resources:

1. Gambas project homepage – (the starting point)

<http://gambas.sourceforge.net/en/main.html>

2. Gambas Wiki Encyclopedia –

<http://gambasdoc.org/help?en>

3. Language references – Note that these same resources (below, in this section) are available offline, for your current version of Gambas distribution, if you click the “?” item on your Gambas IDE menu. If you do that, a browser window will open, giving access to the documentation, without needing to have a Web connection.

<http://www.gambasdoc.org/help/comp>

This is the gateway to documentation of all Gambas2 components and classes – datatypes, controls, properties, methods, etc.

<http://www.gambasdoc.org/help/comp?v3>

This is the same as the above, but for Gambas3.

<http://gambasdoc.org/help/lang>

This is the Language Index for Gambas2. It includes an Overview section and an Index By Name for keywords and functions.

<http://gambasdoc.org/help/lang?v3>

This is the same as the above, but for Gambas3.

4. Context help –

If you are writing code in the Gambas IDE, you can highlight a word and click on the magnifying glass/question mark icon () and the IDE will open a help browser with that specific Gambas language  defined.

5. Special purpose documents –

<http://gambasdoc.org/help/doc>

This contains documents discussing how to do a variety of special-purpose jobs connected with Gambas.

<http://gambasdoc.org/help/doc?v3>

This is the same as the above, but for Gambas3.

6. Gambas-related sites and forums:

<http://gambaslinux.eq2.fr/index.php?lng=en>

This website is bilingual in French and English. Some of the code samples are documented in French, others in English. There are many useful code samples, some complete programs, and links to other forums and blogs. Note the chat function, “Mini Tchat”, where you can ask questions.

[http://www.gambasforum.com/index.php?
PHPSESSID=fccdc6e9e017d1ad95257aa57ac75956&board=2.0](http://www.gambasforum.com/index.php?PHPSESSID=fccdc6e9e017d1ad95257aa57ac75956&board=2.0)

This is a very well-organized Gambas forum, with a lot of participation. Great for finding advice.

http://gambasrad.org/zforum/index_html

Here is another forum, with thousands of topics and code examples, and many participants.

<http://en.wikibooks.org/wiki/Gambas>

This is a wiki that needs more submissions, but it does contain useful information.

http://pigasoftware.a.wiki-site.com/index.php/Gambas_Examples

This is a wiki that discusses techniques and code examples, with a special focus on games.

<http://beginnersguidetogambas.com/>

This is a WordPress blog, intended to support this book by allowing comments or questions related to the book contents. Contents of the DVD referenced herein may be downloaded from here.

Appendix B

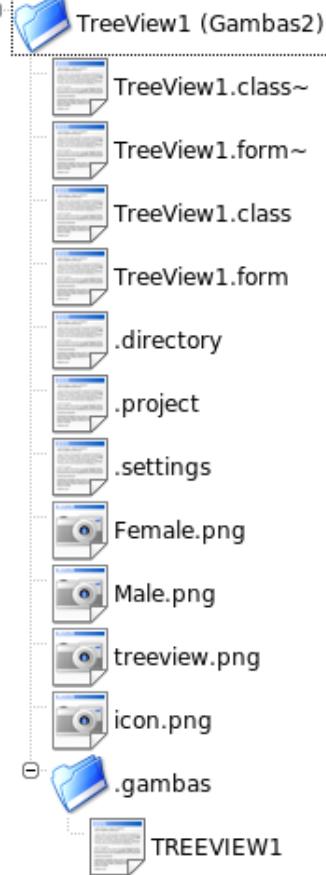
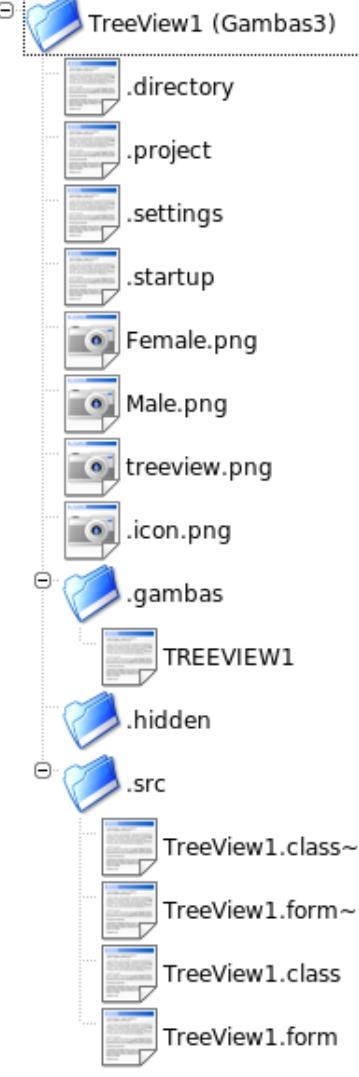
Managing the Gambas experience:

1. Read-only Gambas Examples – In Gambas 2.xx, the Example projects demonstrate a lot of valuable programming techniques: networking, barcodes, charts, games and media, and much more. These can provide the basis for your own great programs, but the Examples included with the installation are “read-only”, so you can't view the GUI properties for the forms and other controls. Solution: the “read-only” property can be removed by simply copying a project folder and pasting it as a new folder in another location. Or, you can copy and paste with a new name in the current location. At any rate, the complete set of Gambas2 Example projects, without the “read-only” property, has been included in the DVD that accompanies this book. See the folder named “Gambas2 examples NOT read-only”.

In the Gambas3 Alpha version, these projects are not, at this time of writing, “read-only”, and we don't know whether or not that will be true in the release version of Gambas3. However, if you encounter a problem, the copy-paste procedure mentioned above will probably solve it.

2. Differences between Gambas 2.xx and Gambas 3.xx – The release date for Gambas 3.xx has not been set yet. There are a number of changes, compared to Gambas 2.xx. If you try to use Gambas3 to open a program written in Gambas2, Gambas will notify you about this discrepancy, and will ask you if you want to convert the project to Gambas3 format. If you reply positively, Gambas will make the conversion. This makes the project incompatible with Gambas2. Therefore, make a backup copy of your Gambas2 project before you make the conversion to Gambas3.

In moving from Gambas2 to Gambas3, some of the changes include simplified syntax, fewer synonyms for keywords and properties, improved consistency, better GTK+ performance, and Qt4 compatibility. There is a major structural change in the project folders, as shown in the following figures:

 <p>TreeView1 (Gambas2)</p> <ul style="list-style-type: none"> TreeView1.class~ TreeView1.form~ TreeView1.class TreeView1.form .directory .project .settings Female.png Male.png treeview.png icon.png .gambas <ul style="list-style-type: none"> TREEVIEW1 	 <p>TreeView1 (Gambas3)</p> <ul style="list-style-type: none"> .directory .project .settings .startup Female.png Male.png treeview.png icon.png .gambas <ul style="list-style-type: none"> TREEVIEW1 .hidden .src <ul style="list-style-type: none"> TreeView1.class~ TreeView1.form~ TreeView1.class TreeView1.form
<p>Figure 1- TreeView1, original project structure under Gambas2</p>	<p>Figure 2- TreeView1 project structure, converted to run under Gambas3</p>

Note that the filenames with tilde characters (~) are backup files that appear after you have edited the original program. Set your file browser to “View hidden files” when you look inside a Gambas project, because most of these files and folders are hidden. Also, these treeviews of TreeViews were generated using the TreeView of Chapter 8. Recursion!

If you convert a project from Gambas2 to Gambas3, there is no guarantee that it will run

correctly. Gambas just converts the project structure, as shown above – it does not attempt to change your code to insure compatibility, and the programmer must troubleshoot any needed changes. Note the webpage:

<http://gambasdoc.org/help/doc/gb2tob3?v3>

The page title is “Porting from Gambas 2 to Gambas 3”, and the table on that page tries to take an inventory of all incompatible changes between the two versions.

3. Changing the project properties after creation – When you start a new Gambas project, you respond to a number of prompts until the project is created. For example, you have an opportunity to include the components for Internationalization, Database access, Network programming, etc. The configuration that you chose at creation is not fixed forever, therefore don't worry if you find later that you need to add something, or that a component was unnecessary. The “Project” menu gives you a “Properties” dialog that allows you to assign a project title (if you didn't do that before), change a variety of options (including translatability) and check or uncheck components that will be used in the project. You can even switch from the Qt toolkit to GTK+, and vice versa. It's a good idea to spend some time exploring the top menu bar to see all of the tools that are available to you.

4. Random notes on project management from the IDE –

- Every time you execute a program from the IDE, the current state of your project will be temporarily saved before execution, and you will be able to undo your changes by clicking on the “back” icon.
- Every time you “Save” your project, your class/form/module files will be updated to the current state, and your previous version is saved into the backup files with names that include tildes (~), as shown in item 2 above. After saving, if you run into a serious problem, you can't use the “back” icon anymore, but you can delete (or rename) your current class/form/module files, then rename the backup files by removing the tildes. This restores your project condition prior to the last “Save”.
- When you execute your program from the IDE, your editing toolbar or Properties tab will disappear, and the debugging toolbar will be active. If the interpreter encounters an error that causes the program to freeze, you must terminate the program before you can resume editing. The square “Stop” icon on the debugging toolbar is the best way to terminate the program and return to editing.