Cover design by Fabien Bodard
Foreword by Fabien Bodard and Benoît Minisini

A Beginner's Guide to Gambas

A Beginner's Guide to Gambas

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.

A Beginner's Guide to Gambas

Table of Contents

A Beginner's Guide to Gambas

A Beginner's Guide to Gambas

Table of Figures

A Beginner's Guide to Gambas

This page is intentionally blank.

Acknowledgments

First of all, a very special thanks to **Benoît Minisini** for the creation of Gambas and for his support of this effort to further document this wonderful language. Without Benoît's initiative, we would all be struggling to find a better tool than what exists today on Linux platforms. Much of the initial documentation of the Gambas language was put on the Gambas Wiki by Benoît and he deserves special credit for making this information available. As it was the only known published source of definitive documentation in existence prior to this writing, much of the reference material herein was gleaned from that initial set of documentation. As with any written material, there is no guarantee that this documentation is as accurate as what you may find in the latest release of the Gambas product.

The author would like to extend a special thanks to **Fabien Bodard** for his help in making this work become a reality. Fabien tirelessly edited code, reviewed the sample projects presented herein, and provided great insight into the inner workings of Gambas – all at the time of harvest for his vineyard. **Laurent Carlier** and **Steve Starr** both also deserve my gratitude for their meticulous work editing the code and ensuring everything worked as advertised. Their feedback, suggestions and corrections were greatly appreciated. I would also like to thank **Daniel Campos** who was a great asset in helping me to understand some of the finer points of Gambas. **Nigel Gerrard** also contributed to the success of this project by providing a final review and edit of the database material. From among these Gambas Hall of Fame coders, who else could have done it better?

Countless emails were received from many members of the Gambas community, all of which provided ongoing encouragement and support which motivated me to complete this effort and make it available to everyone. Given the difficulties of writing a book in the first place, it was a great feeling of satisfaction to receive such support from complete strangers who all share the single vision of making Gambas a success. I can only hope that this book will do justice to that effort and welcome any suggestions for change, as well as any compliments or constructive criticism. It has been a lot of fun to learn this wonderful new language and to explore the possibilities it offers. I dedicate this book to all of those Gambas users and developers who have relied on each other in the Gambas user/developer community to make this product become reality. Shared vision, common goals, and open software are powerful forces that should not ever be underestimated. Such forces have been known to change the world and will continue to do so long after this book has been read and forgotten.

John W. Rittinghouse, Ph.D., CISM
October, 2005

Foreword

For the last three years, I have plunged headfirst into the Linux programming environment. Considering my first installation of Linux was back in 1996, it should not be considered a good first start. As an enthusiastic Basic developer, the Linux environment lacked a tool that would allow me to easily program in the Linux environment. By chance, I stumbled upon a small project, the fruit of more than two years of work from a man named Benoit Minisini. After figuring out how to get past the idiosyncrasies of compilation, I discovered what was to be the embryonic stages of one of the most fabulous projects i could have imagined, Gambas! At that time, Gambas was already implemented with it's own Integrated Development Environment (IDE) and a syntax highlighter. The ability of Gambas to dynamically load components and remain fairly true to the Basic language as used under Windows was an added benefit.

Today, Gambas has arrived as a mature product that, in it's first version, allows a user to construct graphical applications or console based applications, supports database management, connection to the Internet or to socket servers, utilizes data compression, supports DCOP with KDE applications, and much more. A Gambas application can be translated directly from the IDE and packaged in binary form for different Linux distributions. What was initially alluring to me still is -- Gambas has gone beyond being just another programming language because it supports all of the features of a "professional" product while retaining its simplicity. Gambas provides a real tool for the beginning programmer and makes it possible for a novice programmer to develop high quality applications. Here, in this book, you will find the first comprehensive treatment of Gambas. It makes approaching the Gambas language simple and the reader can easily progress from the beginner's level topics to the more advanced topics professional programmers use daily.

Gambas has and will continue to evolve. Gambas Version 1.0 (discussed in this book) is the foundation of a language which will evolve to be even more powerful. For example, Gambas Version 2 will make it possible to make API calls to a native library. It will allow programmers to manage even more types of database servers and will work equally well with either Qt or GTK graphics libraries. Gambas 2 will allow programmers to develop their own components from within the Gambas environment. Developers will be able to create games using SDL and OpenGL. The scope of the Gambas language is growing larger and and the syntax is becoming more compact. All of these ideas for improvement

and change do not rest solely in the heads of a stable of chosen developers working on the Gambas 2 project. They come from all Gambas users who provide the ideas that make Gambas 2 even better than before. Already it can do so much more than version 1.0. I strongly recommend that while you are waiting for the stable release of Gambas 2, you begin your Gambas training with your machine and what you will find in this book. It will prepare you for a wonderful programming experience in the Linux environment.

Fabien Bodard and Benoit Minisini

Chapter 1 - Introducing Gambas

Gambas was initially created by Benoît Minisini, a resident of the suburbs of Paris. According to Benoît, Gambas is a Basic language with object extensions. The name itself is a play on the words "**G**ambas **a**lmost **m**eans **Bas**ic" and, according to the author, Gambas evolved because of his personal programming experiences with the Microsoft Visual Basic® software product[1]. Rather than contend with the horrendous number of bugs and idiosyncrasies found in that product, he decided to create Gambas.

Gambas is licensed under the GNU Public License[2] and has taken on a life of it's own due to its immense popularity. Gambas runs on most of the major Linux platforms and the current (at the time of this writing) stable version is Release 1.0.9. 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 [sic] 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."[3]*

Minisini disliked the overall poor level of programming that is common among many Visual Basic programs. Many believe that this problem may be due to the "enforced" use of quirky programming practices imposed on developers as a result of the wide range of bugs and the strange idiosyncrasies of the proprietary VB language. Gambas has been 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 the Basic programming language has 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 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:

---

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

2   For information about the license, visit http://www.gnu.org/licenses/licenses.html#GPL .

3   See the Gambas Wiki Web Site Introduction at http://gambas.sourceforge.net/index.html for more details.

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

Figure 1 below[4] is an illustration of the overall architecture of Gambas. In Gambas, a project contains class files, forms, modules, and data files. A Gambas project is stored in a single directory. Compiling a project uses in incremental compile method that only requires recompilation of the modified classes. Every external reference of a class is solved dynamically at runtime. 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.
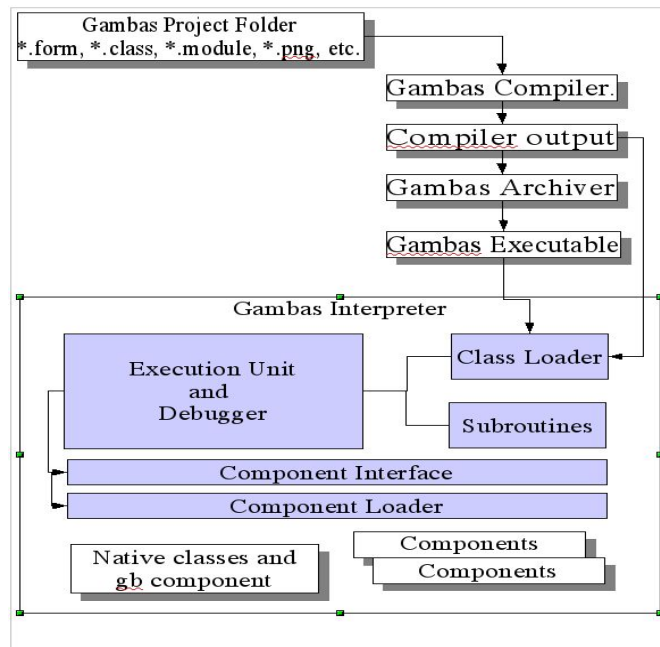
Figure 1- General overview of Gambas architecture.

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[5]***.

---

4   See URL http://gambas.sourceforge.net/architecture.html, for the original graphic.
5   Copyright (c) 1999-2005 by contributing authors. All material on the Gambas Wiki is the property of the contributing

We will cover Gambas components in greater detail later in this book. By default, the Gambas Interpreter is a text-only (console-based) program. The component architecture is used for the graphical user interface (GUI) part 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+[6], Qt[7], 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 the QT library. It is recommended that the reader also consult with the QT documentation[8] to better understand the GUI controls. According to Minisini, a GTK+ component with a nearly identical interface as the Qt component is planned.
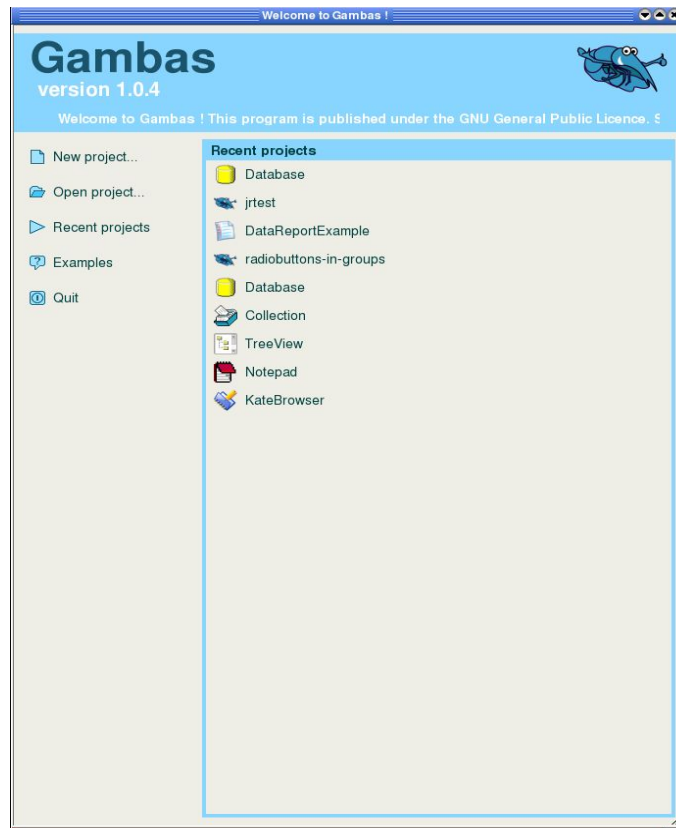
Figure 2- the Gambas Opening Screen.

---

authors.

6   For more info about GTK+, visit http://www.gtk.org/ .

7   For more info about Qt, visit http://www.trolltech.com/products/qt/ .

8   http://doc.trolltech.com/3.3/index.html

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 when we cover Internationalization (i18n).

The Gambas Programming Environment

For now, let's take a quick tour of Gambas and introduce you to the Gambas programming environment. All of the examples and code written for this book were developed using Gambas version 1.0.9 running on Linspire® 5.0[9]. They should work on any platform that you can install Gambas on successfully. For Linspire® users, it is as simple as the click of a button using the Click-N-Run® warehouse feature of that product. When you first click the desktop icon to 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 our first Gambas project. You will see a dialog similar to Figure 3 below appear.

Figure 3- The Gambas Project Creation Wizard.

---

9   For more info about Linspire, visit http://www.linspire.com .

Simply click the **Next>>** button and you will be presented with another dialog that requires you to choose the type of project you want to create (see Figure 4, below). This dialog requires you to pick between a GUI project or a console project.
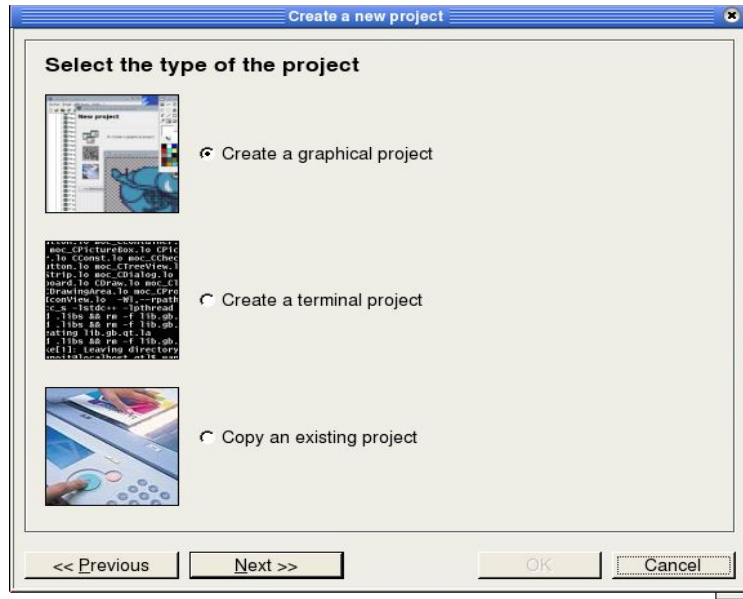
Figure 4- A dialog to select the type of Gambas project you wish to create.

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 will see requires you to name your project and choose where it will be located on your local file system. It is shown in Figure 5 below.

This section of the wizard has you to specify the name of the project as it will be stored on disk. Remember, in Gambas a project is entirely self-contained in a directory or folder. The folder name you choose as the name for the project is what will be created on your system.

The title of the project will be what you specify in the second text input field of the dialog. Additionally, the Options section allows you to choose whether or not your project will be translatable and if the controls used on your forms are going to be made publicly accessible. We will discuss these option in more detail later in this book. For now, just fill out the dialog as shown in Figure 5 and click **Next>>**.
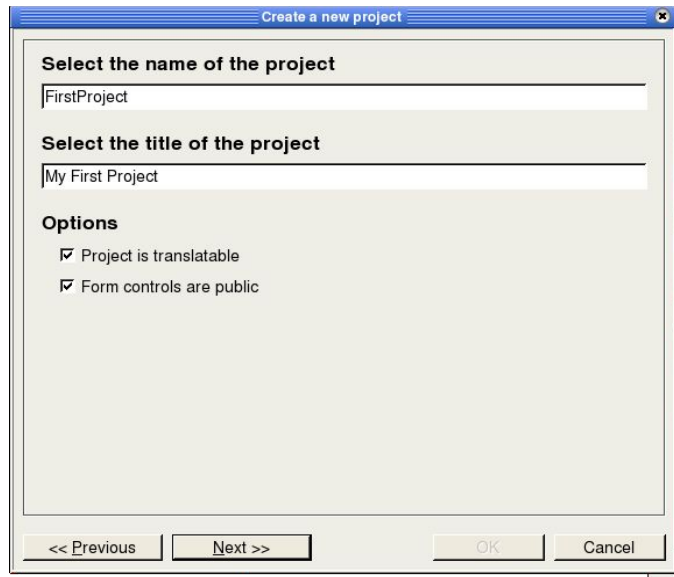
Figure 5- Project Name Selection Dialog.

Choose a directory appropriate for your file system and click the **Next>>** button to proceed to the final dialog screen of the wizard, as shown on the following page in Figure 7. This dialog is simply a confirmation screen of the choices you have made. 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.
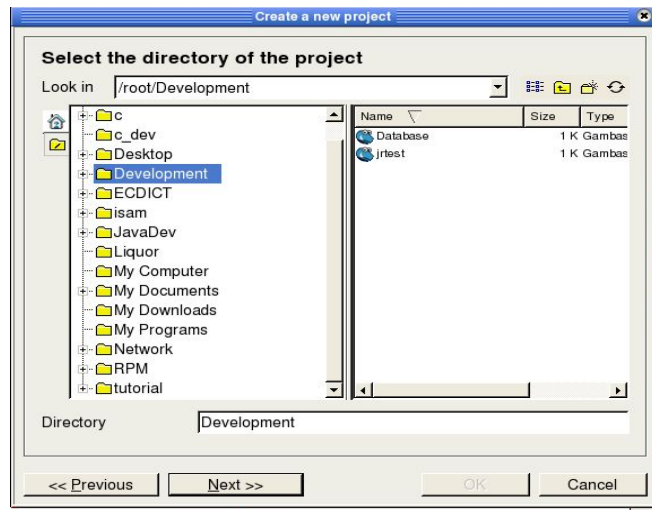
Figure 6- Project Directory Selection Dialog.
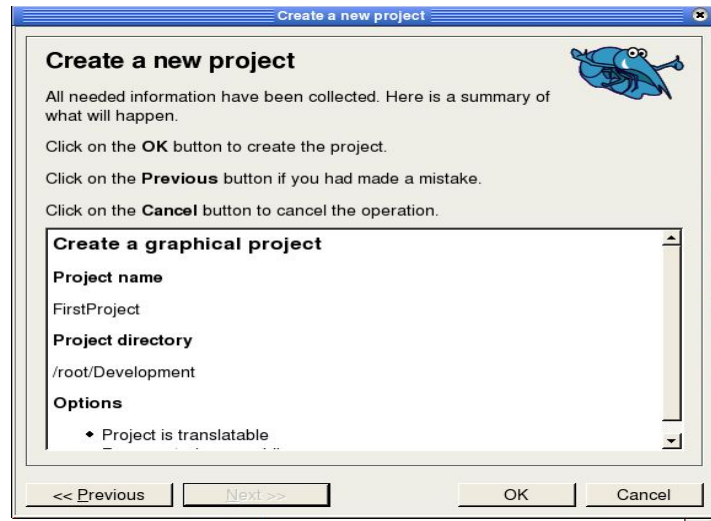
Figure 7- New Project confirmation dialog.

Once you have clicked on the "**OK**" button to finish this wizard, Gambas will present you with the Gambas integrated development environment (IDE). The IDE consists of several components that are shown in Figure 8 on the next page.

Gambas IDE Components

The **Project Explorer** is the main Gambas window. It show 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)* and 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:

- ✔ Classes
- ✔ Forms
- ✔ Modules
- ✔ Data

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 runtime, with code to define properties, methods and event handlers for each object you create.
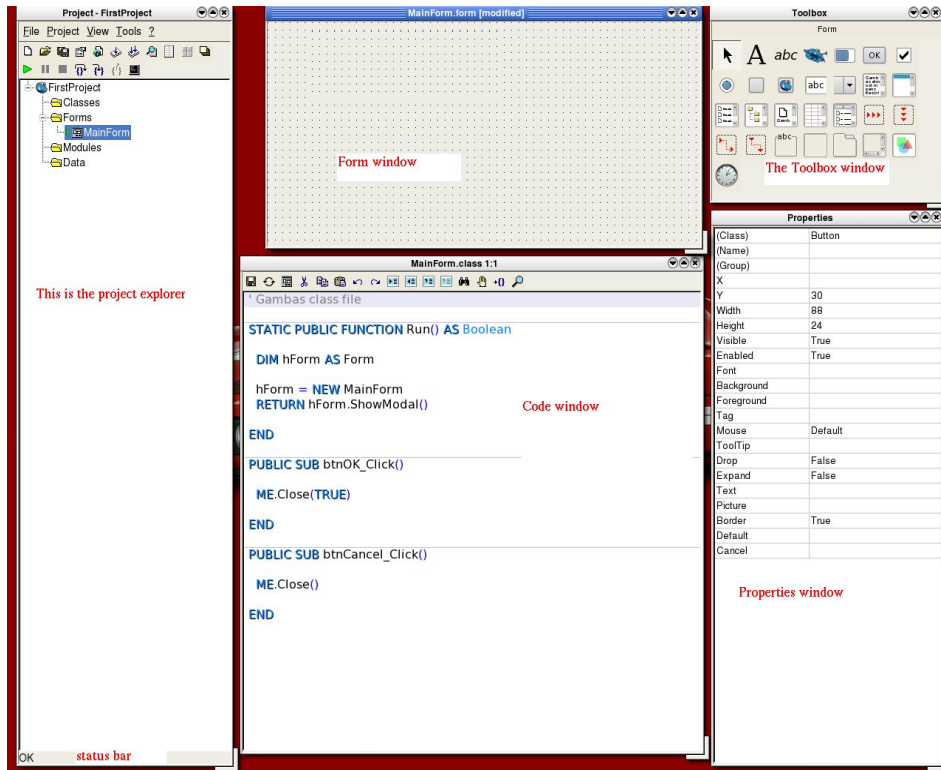
Figure 8- The Gambas IDE.

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

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 runtime and they have no event handlers.

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. 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 interaction between the user and the GUI is referred to as events. The controls are found in the ToolBox window. You can change the appearance and behavior of the controls by setting the properties for each control. Properties can be seen in the properties window.

Now, lets take a look at the menus and buttons found at the top of the Project Explorer. The menus *(see Figures 9 through 12 below)* control all the main Gambas management tasks. The File menu will allow you to open a project, save a project, create a new project, open some Gambas example projects, or quit using Gambas. The Project Menu is where program compilation occurs. You can also create the program executable, make a source archive, or create an installation package. This menu offers you the option of translating your program into another language. Finally, you can set Gambas IDE properties from this menu.
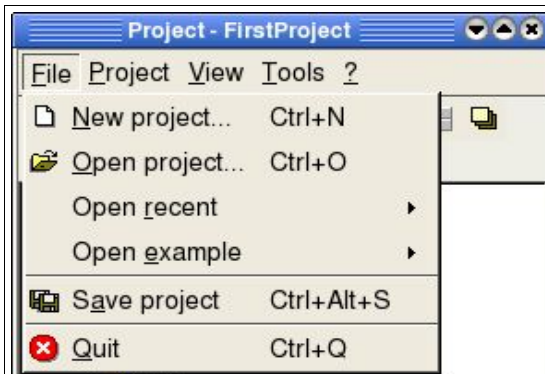
Figure 9- Project Explorer File Menu.
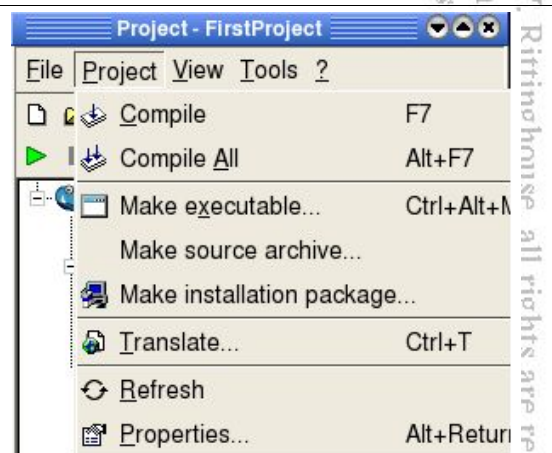
Figure 10- Project Menu.

Figure 11- Project Explorer View Menu.
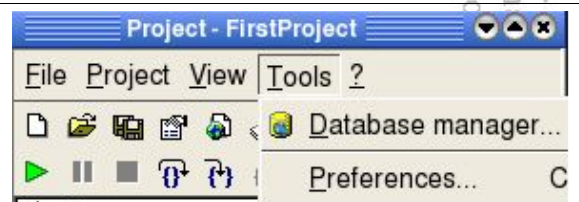
Figure 12- Project Explorer Tools Menu.

The View menu will allow you to bring up the properties window or the ToolBox window. You can open a console or use the Icon Editor to create program icons. The Hierarchy option will open a window and show you your project's class hierarchy. Finally, from this menu, you can close all of the windows that are currently open in the IDE. The ToolBar buttons *(see Figure 13 below)* 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 13 below shows you the File menu and ToolBar. From the Project Explorer TreeView you can double-click on a form and it will pop up for you to edit. 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) it on your form.
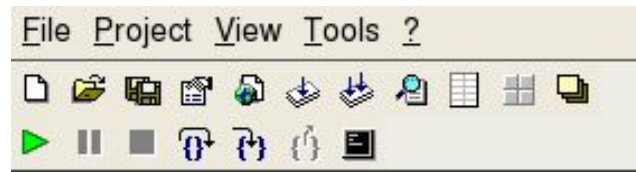
Figure 13- Project Explorer Menu and Toolbar.

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.

From the Project Explorer File Menu, choose the Quit option and save your project. When we reopen it, all your work will be saved. In the next section, we will begin to cover the essentials you need to understand in order to program 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 Gambas coding environment, the primary language concepts that are required to use Gambas, and some basics about data-types and variables. That is all covered in the next section and constitutes the remainder of this introduction to Gambas programming. For now, take a short break and we will continue when you come back.

Chapter 2 – Gambas 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.

<u>Gambas Variables, Data-types and Constants</u>

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 nine basic data-types a programmer can use to write program code in Gambas[10]. These data-types include the following:

```
Boolean        Byte        Short
Integer        Float       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 from of the variable name.

Good programming convention encourages this practice, known as the Hungarian Notation[11], 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*.

---

10 In the second release of Gambas (Gambas2) "Long" and "Single" data types are planned. These data-types are like the C LONG LONG and FLOAT data types. This is a significant improvement because Gambas2 will provide direct programming access to the C API and will also support 64 bit platforms.

11 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.

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. This is similar to the LONG data-type used in VB. 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-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
```

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 table shown below is presented as a convenient reference:

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	4 bytes	0
Float	Similar to the double data-type in C	8 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

Now that you know about all the different types of data that Gambas supports, we will start to look at what you can do with those data-types. 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. 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 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.	&H80A0&, &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 comparison operators then take a look at arithmetic operators and string operators.

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 ...

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.

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 Project Explorer File Menu select New Project. As you go through the New Project Wizard, select a Terminal Project and click Next. Name this project *TerminalTest* and place it in a directory called *TerminalTest*. Don't worry about any of the other options. Just click <u>Next>></u> 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. From the Project Explorer find the TreeView item called **Classes** and right-click the mouse. Choose the New... option and take the default name *Class1*. The code window will appear and inside the window it should look something like this:

```
' Gambas class file
STATIC PUBLIC SUB Main()

END
```

Let's take a look at some 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 the functionality of 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 class file** line (*note that comments in Gambas start with the ' [aka the tick mark])* and between the **STATIC PUBLIC SUB Main()** and **END** statements. Once you have entered the code below in the Gambas code window, click the green ▶ button from the Project Explorer ToolBar to execute your program. Here is the code you want to try first:

```
STATIC PUBLIC SUB Main()
   DIM N AS Integer
   DIM R AS Integer
   N = 3
   R = -6
   PRINT "===> ";N;" | ";R; " and "; -N ; " | "; -R;
END
```

If all goes well (*and it should*), you will see the blue shrimp (*a.k.a., the Gambas mascot*) dance and the console window will respond with the following:

```
===> 3 | -6 and -6 | 3
```

Note that the value of variable N changed from a positive 3 to -3 and the value of -6 changed to a positive value of 6. 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.

```
STATIC 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:

```
STATIC 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:

```
STATIC 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:

```
STATIC 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
```

A Beginner's Guide to Gambas

If you use a backslash to divide numbers, i.e., **Number \ Number** Gambas computes the **quotient** of the two numbers. 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:

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

```
STATIC 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:
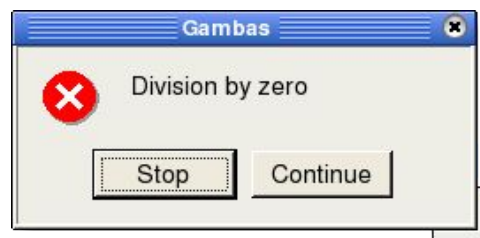
Figure 14- A Division by Zero Error Dialog.

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:

```
STATIC 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 operations[12]. Using the format of **Number AND Number** instructs Gambas to use the **AND** operator to compute the mathematical **AND** of the binary values of both of the numbers.   Try this:

```
STATIC 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
```

Likewise, **Number OR Number** used the **OR** operator and computes the mathematical **OR** of the binary value of the two numbers.

---

12 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++.

```
STATIC 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
```

Number XOR Number uses the **XOR** operator and computes the mathematical exclusive OR of the binary value of the two numbers.

```
STATIC 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
```

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.

```
STATIC 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 *N* number of any type of character.
?	Match any single character.
[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.

```
STATIC 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:

```
STATIC 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
```

Note: you must use a double backslash character, \\ to print a backslash or special string sequence containing backslashes. Otherwise the \ will be interpreted by the compiler as a special character like '\n', '\t', '*', etc. Alternatively, you can use this pattern string: **LIKE "G[Aa][*]"**

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

<div align="center">

Result = NOT Expression

</div>

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:

```
STATIC 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. 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 the first string is lower or equal to the second string
String >= String	Determine if the first string is greater or equal to the second string

Note : all string operator comparisons are case sensitive.

Here are some things to try using Gambas console:

```
STATIC 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
```

45

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. The following table shows all of the Gambas keywords currently supported:

BREAK	**CASE**	**CONTINUE**	**DEFAULT**	**DO**	**ELSE**
END	**ENDIF**	**END**	**SELECT**	**END**	**WITH**
FOR	**FOR EACH**	**GOTO**	**IF**	**LOOP**	**NEXT**
QUIT	**REPEAT**	**RETURN**	**SELECT**	**STEP**	**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 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 ...] [(;|,)]
```

Here is something to try using PRINT:

```
STATIC PUBLIC SUB Main()
   DIM b AS Integer

   b = 1
   PRINT "===> " & "b is: " & B
   PRINT "===> ", " b is:", B
   PRINT "===> "; "b is: " & B
END
```

The console displays the following output:

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

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:

```
STATIC 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
```

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:

```
STATIC 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 to **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:

```
STATIC 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:

```
STATIC 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:

```
STATIC 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
 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:

```
STATIC 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
```

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 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:

```
STATIC 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 & " ==> ";
       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
```

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 in 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. The statements above create a new collection object. Enter the following code into your console code window and run it.

```
STATIC PUBLIC SUB Main()

DIM MyDict AS NEW Collection
DIM strElement AS String

MyDict["absolute"] = 3
```

```
MyDict["basic"] = 1
MyDict["carpet"] = 2

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

Your console should display this output:

```
3 1 2
```

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 cannot wait to jump in, go ahead and turn the page!

Chapter 4 – Introducing the Gambas ToolBox

The default Gambas ToolBox consists of many controls. As of this writing using release (1.0.9), the following controls are currently supported:

- Button
- Label
- TextLabel
- TextBox
- TextArea
- CheckBox
- ComboBox
- ListBox
- RadioButton
- ToggleButton
- Frame
- Panel
- TabStrip
- ProgressBar
- Image
- Timer
- Dial
- SpinBox
- ScrollBar
- Slider
- LCDNumber
- ListView
- TreeView
- IconView
- GridView
- ColumnView
- ScrollView
- DrawingArea
- GambasEditor
- TableView
- Workspace

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 the various functionality the **gb.qt component** provides:

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

We will cover other components later in this book. At the top of Figure 15, you see the word Form displayed. This is the default toolkit for the gb.qt component. For now, let's take a quick look at the ToolBox again and review the icons therein.
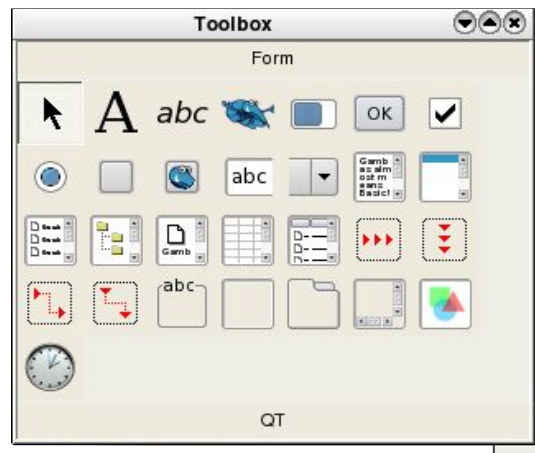
Figure 15- The Gambas ToolBox.

A Beginner's Guide to Gambas

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

- ✔ Select
- ✔ Label
- ✔ TextLabel
- ✔ PictureBox
- ✔ ProgressBar
- ✔ Button
- ✔ CheckBox

On the second row you will find:

- ✔ RadioButton
- ✔ ToggleButton
- ✔ ToolButton
- ✔ TextBox
- ✔ ComboBox
- ✔ TextArea
- ✔ ListBox

On the third row we have:

- ✔ ListView
- ✔ TreeView
- ✔ IconView
- ✔ GridView
- ✔ ColumnView
- ✔ HBox
- ✔ VBox

The fourth row of controls display:

- ✔ HPanel
- ✔ VPanel
- ✔ Frame
- ✔ Panel
- ✔ TabStrip
- ✔ ScrollView
- ✔ DrawingArea
- ✔ Timer

At the bottom of the ToolBox window, the word QT is seen. If you click on that, you will see the controls specific to the QT user interface appear, similar to Figure 2 below:
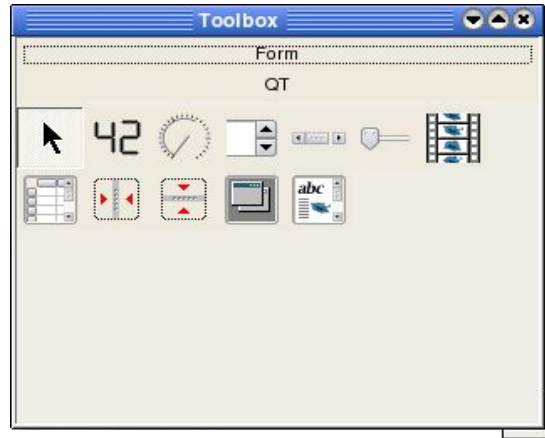
Figure 16- Additional controls for QT.

The QT-Specific Controls found on the top row of Figure 2 are:

- ✔ Selector
- ✔ LCDNumber
- ✔ Dial
- ✔ SpinBox
- ✔ ScrollBar
- ✔ Slider
- ✔ MovieBox

The second row of controls include these tools:

- ✔ TableView
- ✔ HSplit
- ✔ VSplit
- ✔ Workspace
- ✔ TextView

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. 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. 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.

BackColor is defined as **PROPERTY BackColor AS Integer**

This integer value represents the color used for the control background. It is synonymous with the Background property. 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 BackColor property for hButton to red, you would use this code:

```
hButton.BackColor = 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 BackColor (*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.BackColor = 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. Start Gambas and open up the project we created named FirstProject. From the TreeView, double-click your mouse cursor on the MainForm 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()
  MainForm.Close
END
```

Once you have entered the code above, click the green program run button and you should see something similar to the following:

Figure 17- First Button code results.

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. Remember, a good programmer will always try use predefined constants if they are available instead of hard-coding numeric values. 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 anticpated gb.gtk component will not work with gb.qt code if you are not using predefined constants so when writing code, it is imperative to use these constants instead of numeric values. If not, the code will not be portable. Good code should never use numeric values for these tasks. Now, save your project and exit Gambas for now and we will proceed with learning more about the ToolBox.

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 P**ROPERTY 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 it's 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 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 18- Demonstrating font capabilities.

ForeColor is defined as **PROPERTY ForeColor 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 ForeColor property for hButton to red, you would use this code:

```
hButton.ForeColor = Color.Red
```

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

```
hButton.ForeColor = 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
Default	-1
Arrow	0
Cross	2
Wait	3
Text	4
SizeS	5
SizeE	6
SizeNESW	7
SizeNWSE	8
SizeAll	9
Blank	10
SplitV	11
SplitH	12
Pointing	13

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 it's 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 to read as follows:

```
PUBLIC SUB hButton_Click()
  Button1.Hide
  Label1.Text = "Ouch!"
  WAIT 2.0
  Button1.Show
  Label1.Text = "Better now!"
END
```

Once you have entered the code, run the program and see what happens. When you click the button it will disappear and Label1.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 19- The dotted line indicates focus for a control.

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. Close Gambas (*a small quirk of Gambas is the lack of a close project menu selection*) and restart it to initiate the New Project Wizard. Select a graphical user interface project and go through the wizard just like we did with the project FirstProject. Name this project SecondProject. Once the IDE appears, create a new startup class form and name it Form1. For this project, we are going to place three Label controls on the form, as shown in Figure 20 below.
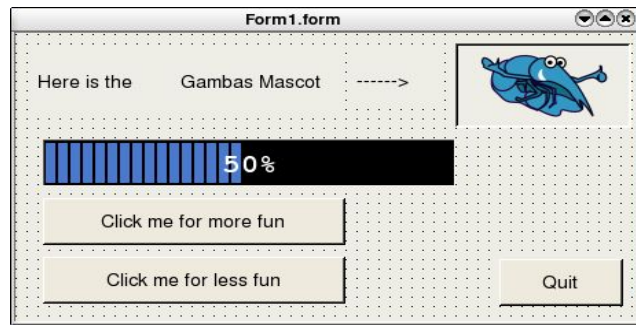
Figure 20- The layout for SecondProject Form1.form.

The first label's text caption is set with the Label.Text property. The setting should be "Here is the" and the second is "Gambas Mascot" and the third is "----->" which points to a PictureBox control (where the shrimp lives). Name the labels **Label1**, **Label2** and **Label3** respectively. Next, name the PictureBox control **PictureBox1**.

To obtain the graphic of the Gambas mascot, you can go to the official Gambas web site[13] and right click on the mascot. Choose to save the image and samve it as "gambas mascot.png" on your computer. Move the file to the data directory in the SecondProject folder. *(Another 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 this:
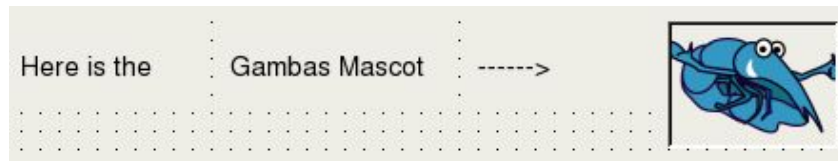
Figure 21-A partially constructed form with our first four controls.

Now, select the **ProgressBar** control from the ToolBox and place it on the form so it looks like that shown in Figure 20 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 Figure 3 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 (**Form1.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 zero when the program starts. That is handled with the initialization routine when opening the form at program runtime, as shown below:

---

13 The official Gambas site is http://gambas.sourceforge.net/index.html .

```
' Gambas class file

PUBLIC SUB Form_Open()
   ProgressBar1.Value = 0.50
   ProgressBar1.BackColor = Black
   ProgressBar1.ForeColor = Yellow
END
```

When the mouse moves over the startup form (*our Gambas program on the desktop*), we want to set the three top label's 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 Figure 22 below. 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
```
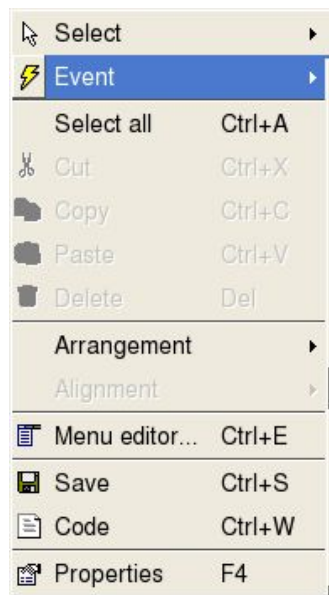

Figure 22- Event menu.

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:

A Beginner's Guide to Gambas

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

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's 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.Plain
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
```
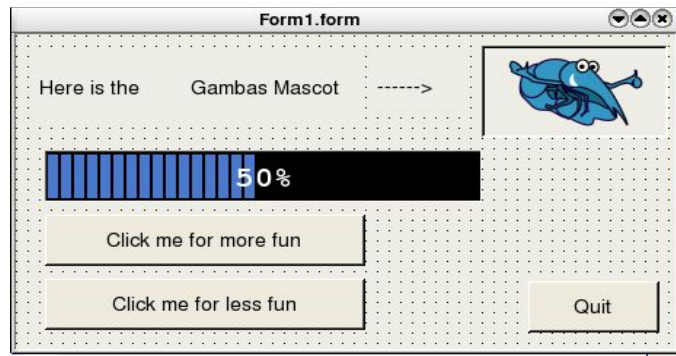

Figure 23- Adding the FunBtn to our form.

Now, we have to write code for the ProgressBar. For our example, we will increment it 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
```

Now, 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
```

That is nearly all there is left to do! Now, save the project and click the run button. You should see this on program start:
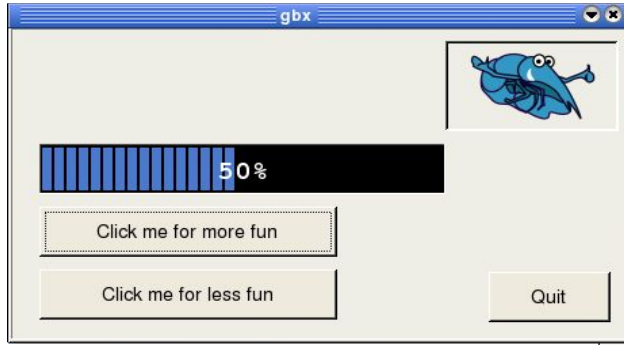
Figure 24-What our form looks like when a mouse is detected over the form. Note the text is blanked out.

Now, click the "Click me for more fun" button and the progress bar changes. Click it three times and you should see this:
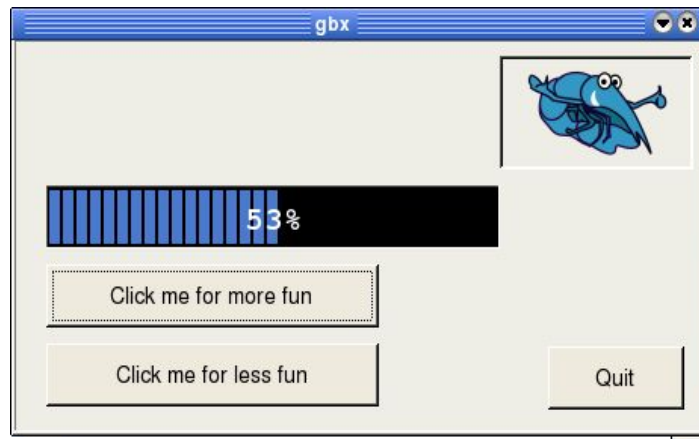
Figure 25- The progress bar when the FunBtn is clicked three times.

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 the other events that can 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 it's 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 Gathering 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:
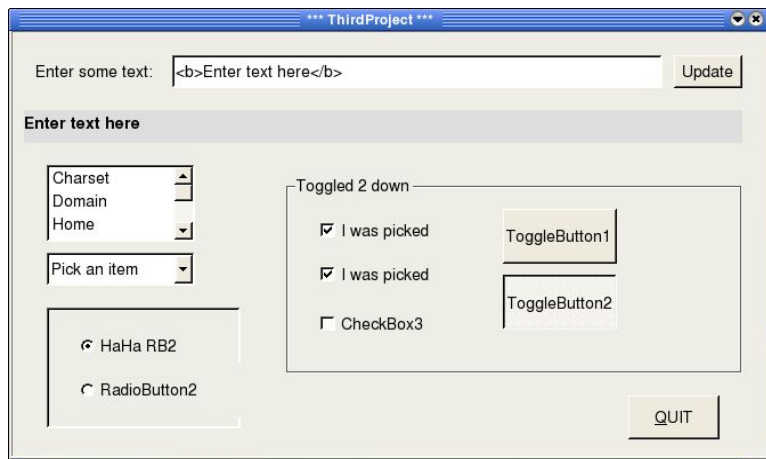
Figure 26- ThirdProject (almost) final result.

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 Graphical User Interface. Go through the wizard making the project translatable and the form controls public. Select the name **ThirdProject** and put it in a directory of your choosing. When the IDE starts, create a new, startup class form. 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 = " *** ThirdProject *** "
```

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()
  Form1.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. It implements a control capable of displaying simple HTML text. You can use HTML entities such as **&lt** and **&gt** to display characters like **<** and **>**. For HTML that requires the use of quote marks, avoid the " " because it will not work. For example, entering:

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

The figure below shows you an example of how this works:
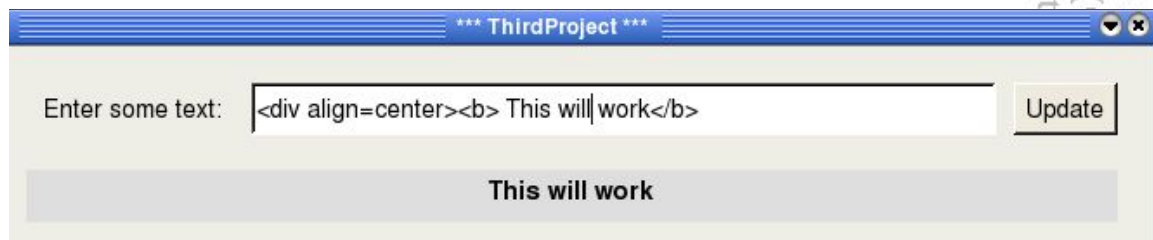
Figure 27- Using HTML to format TextLabel output.

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 cool 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> <b>" & TextBox1.Text & "</b>"
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 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 Figure 28 below:
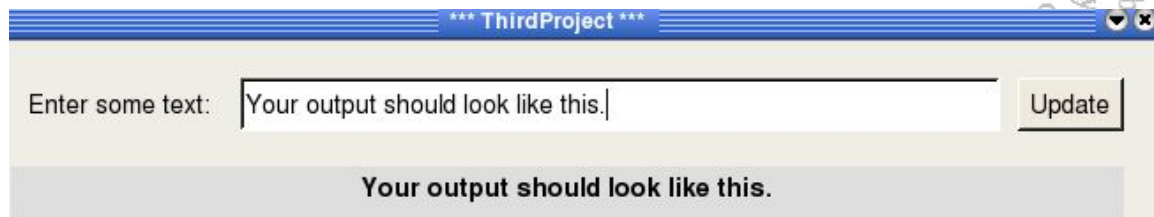
Figure 28- Modified TextLabel output using HTML formatting.

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 now. 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 (**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 the user would 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?
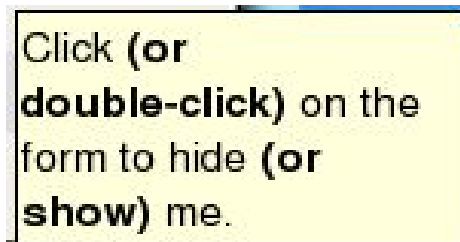
Figure 29- Adding a ToolTip to inform the user how to show/hide a control.

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 30- Our ComboBox.

OK, i agree that it is nothing 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. 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:

Figure 31- 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 Figure 31 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:

Item 3 was selected.
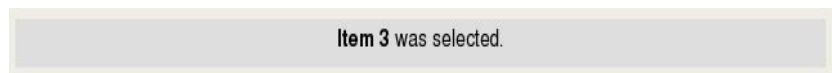
Figure 32- Formatting a TextLabel with HTML.

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:

Pick an Item + -

Figure 33- 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. It 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 34- 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:
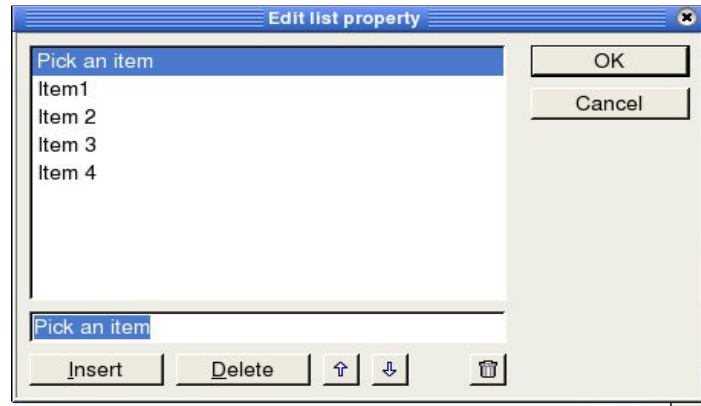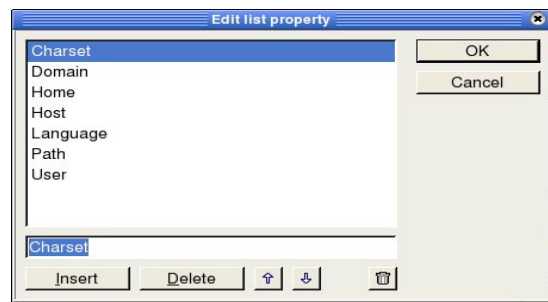

Figure 35- ListBox Edit list property editor.

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 **componen**t library and provides support for all the classes included in the interpreter by default. 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
> Home
> 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) = "Home" THEN
    TextLabel1.Text = System.Home
  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 = System.User
  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:
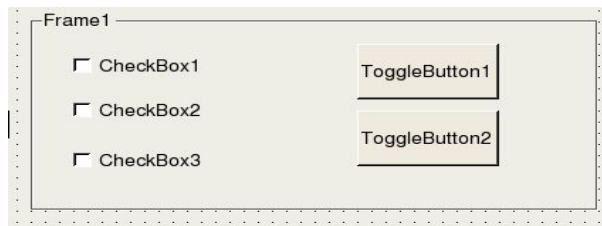
Figure 36- What the example frame we build will look like.

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 Figure 24 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 Figure 33 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()
 DIM result AS String
  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()
  DIM outline1 AS String
  DIM outline2 AS String
  DIM outline3 AS String

  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 )
```

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 37- A Panel with RadioButtons.

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 Figure 37 above.

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. This class is creatable. Declare and instantiate a 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 RB1"
  RadioButton1.Text = "RadioButton1"
END
```

Now, save your project and run the program. 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:
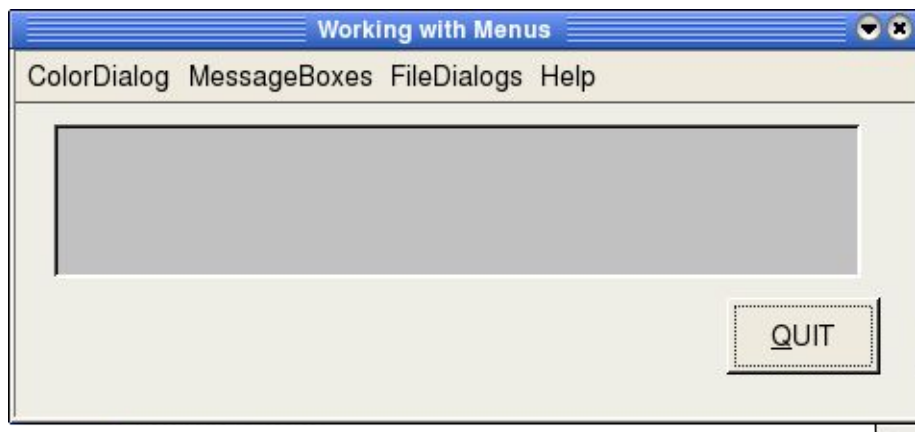
Figure 38- Menu Project Final results.

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 create a new Graphical User Interface project. Name it MenuProject and set all the controls translatable and public. Once you get to the IDE, create a form, Form1 and make it a Startup class. 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()
 Form1.Close
END
```

Double-click on the form anywhere outside a control boundary and you will get the **PUBLIC SUB Form_Open()** subroutine to appear in the code editor. Insert this code in that empty subroutine:

```
ME.Caption = " Working with Menus "
```

Next, add a TextLabel, named TextLabel1 to the window as shown in Figure 38 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

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 and you will see something like this:
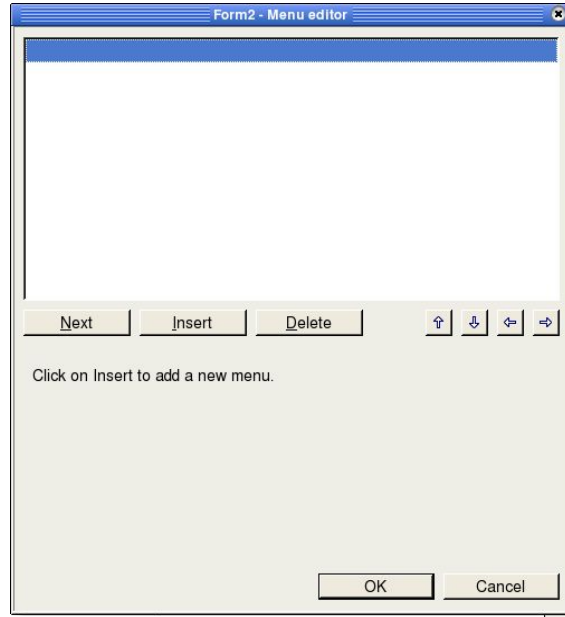
Figure 39- The Gambas Menu Editor when it first starts.
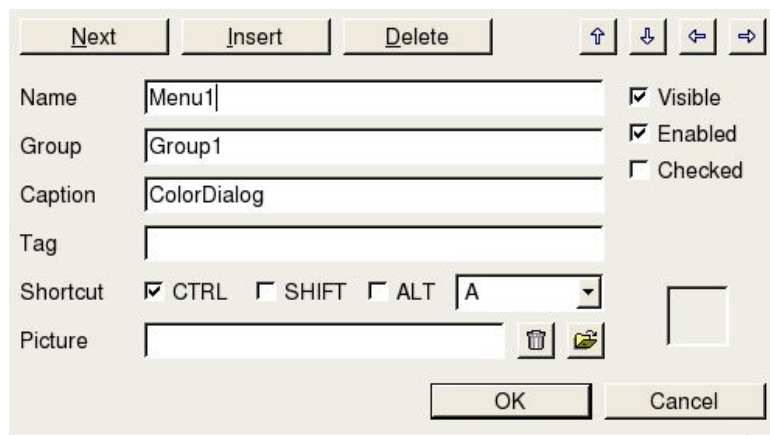
When you first start the Menu Editor, the editor window will, of course, be blank. It is not an intuitively obvious tool to use, so let's spend a minute to explain the Menu Editor. First, some common terminology. The text that you see running horizontally across the top of a form is referred to as a 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.

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 ↑ and ↓ 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 ← or → (un-indent and indent) arrow buttons, the 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 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. Once you click the insert button, you will see several new fields appear at the bottom of the Menu Editor, namely these:

Figure 40- Edit fields for 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 runtime, 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.

NOTE: 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.

Building Menus

Now, let's create our menu. When you are finished, it will look something like this from the menu editor:
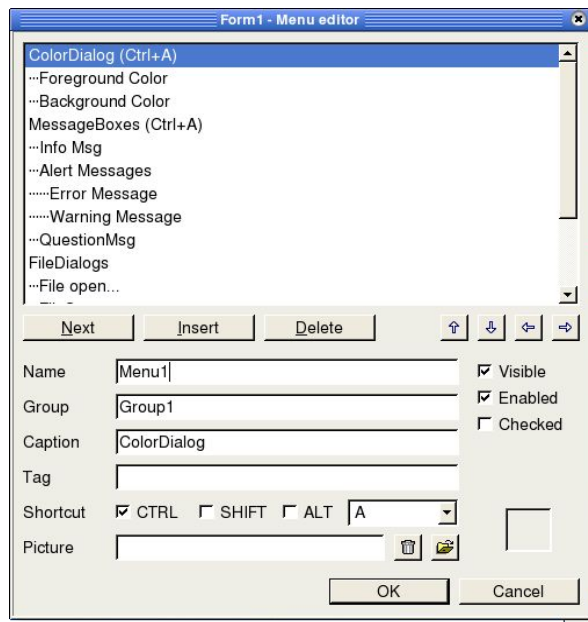
Figure 41- Building our project menu.

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

table 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

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. 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 your 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, OpenFile Dialog and SaveFile Dialog) 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 Menu1Item1_Click()** where you will want to insert this code:

```
PUBLIC SUB Menu1Item1_Click()
  Dialog.Title = "Choose a foreground color"
  Dialog.SelectColor
  TextLabel1.ForeColor = 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:
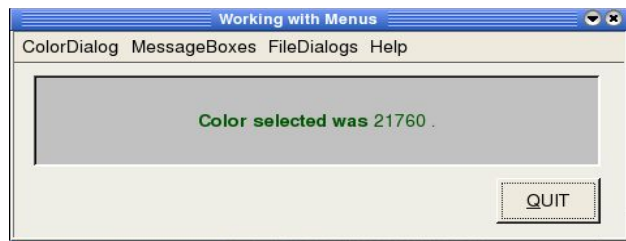
Figure 42- A formatted text label displaying the color value.

Well, 21760 is not very useful information. This number is the internal representation of that greenish color I 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 were 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 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. This module will that 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 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:

```
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
```

```
   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
```

Once you have completed this code, save this module file and return back to the code window for Form1. We have to modify our code in the Menu1Item1_Click subroutine. Here is the new subroutine:

```
PUBLIC SUB Menu1Item1_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 Form1.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)
```

and it will return the string 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 = "<b>Color selected was</b> " & ColorName & "."
```

The final version of our Menu1Item1_Click routine should look like this:

```
PUBLIC SUB Menu1Item1_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 Menu1Item2_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
```

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. To declare a font variable, use 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 Menu1Item3_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 Menu1Item3_Click()
 DIM fontdata AS font
```

```
DIM oldfontdata AS Font
DIM attr AS String
DIM sel AS String
```

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:
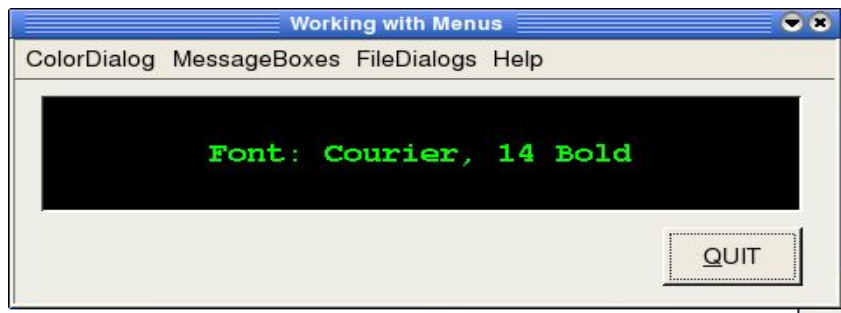
Figure 43- Selecting colors and fonts.

And if you decide to keep it as the default, you will see something like this:
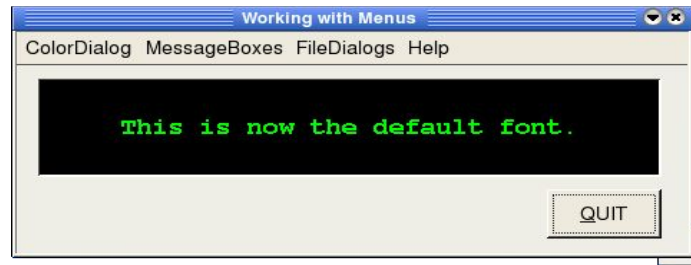
Figure 44- Making a new default font for the TextLabel1 control.

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 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:
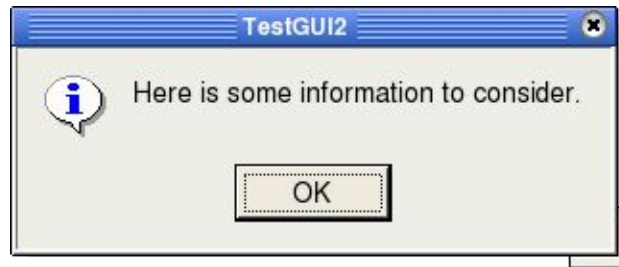
Figure 45- An Information MessageBox.

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 **Menu2Item1_Clicked()** subroutine. Here it is:

```
PUBLIC SUB Menu2Item1_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:

```
PUBLIC SUB Menu2Item1_Click()
 IF Menu2Item1.Checked THEN
    Menu2Item1.Checked = FALSE
 ELSE
    Menu2Item1.Checked = TRUE
 ENDIF
 Message.Info("Here is some information to consider.")
END
```
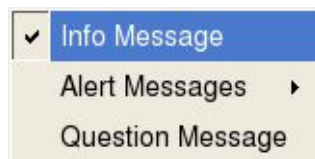

Figure 46- A checked menu item.

Figure 43 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
        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
```

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:

```
PUBLIC SUB Menu2Item3_Click()

  SELECT Message.Question("Did you like this?", "Yes","No","Don't know")
  CASE 1
      TextLabel1.Text = "Liked it."
  CASE 2
      TextLabel1.Text = "Did not like it."
  CASE 3
      TextLabel1.Text = "Did not know."
  END SELECT
END
```

When you execute the program, here is what you should see:
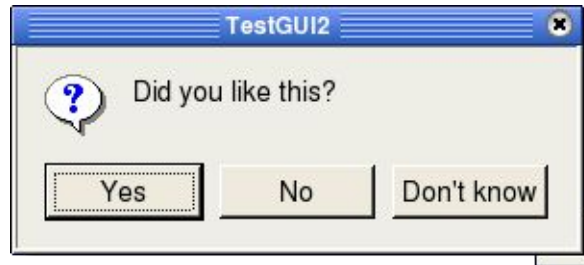
Figure 47- A Question MessageBox.

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:
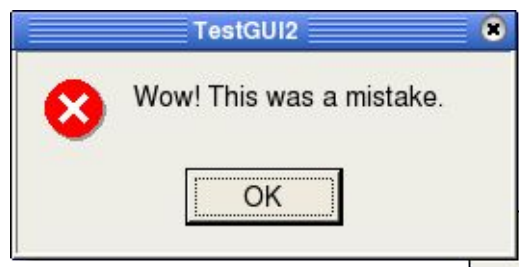
Figure 48- An Error Message.

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:
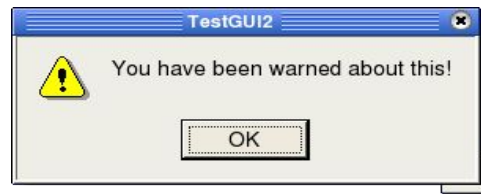
Figure 49- A Warning message.

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:
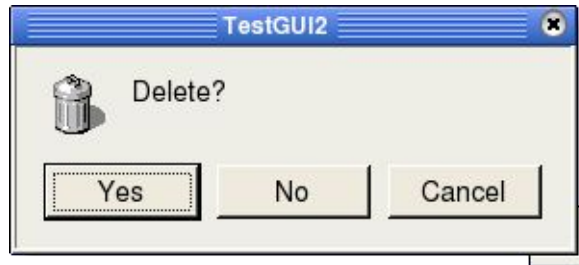
Figure 50- Delete message with three buttons.

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 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:
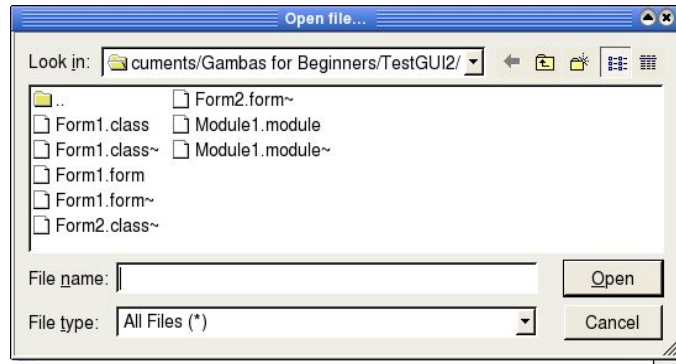
Figure 51- The Open File Dialog.

Pretty easy stuff. Saving files is just as easy.

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
```

Here is the result:
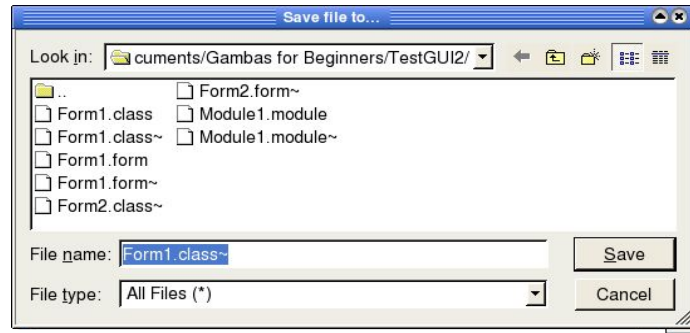
Figure 52- Save File Dialog.

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 Menu3Item3_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:
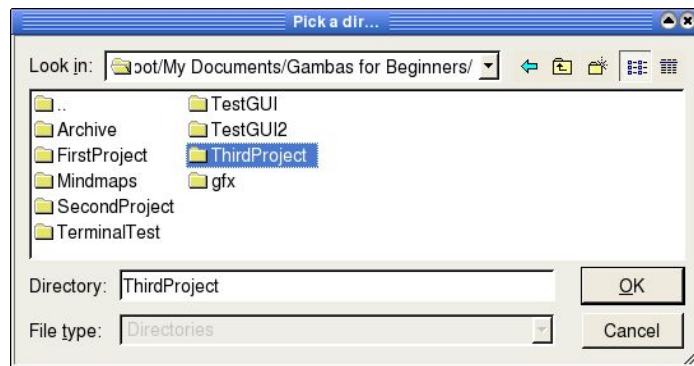
Figure 53- The Select Directory Dialog.

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.

<u>Complete Example Listing</u>

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

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

PUBLIC SUB QuitBtn_Click()
 Form1.Close
END

PUBLIC SUB Menu1Item1_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 Menu1Item2_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 Menu1Item3_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 Menu2Item1_Click()
 IF Menu2Item1.Checked THEN
    Menu2Item1.Checked = FALSE
 ELSE
    Menu2Item1.Checked = TRUE
 ENDIF
 Message.Info("Here is some information to consider.")
END

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

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
           TextLabel1.Text = "Deleted it"
  CASE 2
           TextLabel1.Text = "Not Deleted"
  CASE 3
           TextLabel1.Text = "Cancelled!"
 END SELECT
END

PUBLIC SUB Menu2Item3_Click()
 SELECT Message.Question("Did you like this?", "Yes","No","Don't know")
  CASE 1
           TextLabel1.Text = "Liked it."
  CASE 2
           TextLabel1.Text = "Did not like it."
  CASE 3
           TextLabel1.Text = "Did not know."
 END SELECT
END

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

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

PUBLIC SUB Menu3Item3_Click()
  Dialog.Title = " Pick a dir... "
  Dialog.SelectDirectory
  TextLabel1.Text = "SAVE to DIR: " & Dialog.Path
END

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

Module1.module listing

```
' Gambas module file
PUBLIC FUNCTION SetColorName ( iVal AS Integer) AS String
  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
```

Chapter 7 – Handling Strings and Converting Data-types

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, create a new startup class named Class1. The code window should appear and you should see something like this:

```
' Gambas class file

STATIC 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:

```
STATIC PUBLIC SUB Main()
 DIM sStringLength AS Integer
 DIM sTestString AS String

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

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

 sTestString = "12345678901"
 sStringLength = Len(sTestString)
 PRINT "==> " & sStringLength & " 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 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 and Lower$/Lcase$/Lcase

Upper$ returns a string converted to upper case. Ucase$ and Ucase are

synonynms 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. Here is the standard Gambas language syntax for using these functions:

```
Result = Upper$ ( String )

Result = UCase$ ( String )
```

Type this in the code window and execute the program:

```
STATIC PUBLIC SUB Main()
 DIM sStringLength AS Integer
 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 stricly 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:

```
STATIC 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 14 characters long
<abcdef> is the result of Trim$ call.
```

Left$, Mid$, and Right$

Left$ **r**eturns 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:

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

 PRINT "         1         2"
 PRINT "12345678901234567890"
 sTestString = "abcdefghi"
 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
abcdefghi 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:

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

 PRINT "          1         2"
 PRINT "123456789012345678901234567890123456"
 sTestString = "a"
 PRINT sTestString & Space$(24) & "z"
END
```

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:

```
STATIC 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:

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

 PRINT "          1          2"
 PRINT "1234567890123456789 0123456"
 sTestString = "a"

 PRINT sTestString & String(24,".") & "z"
END
```

The console responds with:
```
         1         2
1234567890123456789 0123456
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 very useful when you must concatenate strings that must be translated. Do not use the & operator, as the order of concatenation might change with the language. Standard Gambas language syntax is:

```
Result = Subst ( Pattern , ReplaceString [, ReplaceString ])
```

Try this little application on your console:

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

 sTestString = "abcdef"
 sResult = "ghijkl"
 PRINT "Our start string is: " & sTestString
 PRINT Subst$("Our Subst string is: &1",sResult)
 END
```

The console responds with:

```
Our start string is: abcdef
Our Subst string is: ghijkl
```

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 class file

STATIC 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)
        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
123456789012345678901234567890123456789012345678901234567890
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. 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:

```
STATIC 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:

```
STATIC 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!
```

Converting Data-types

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 class file
```

```
STATIC 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.

```
String = Bin$ ( Number [ , Digits ] )
```

Example code you can try on the console:

```
STATIC 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:

```
STATIC 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:

```
STATIC 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:

```
STATIC 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:

```
STATIC 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
```

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:

```
STATIC 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:

```
STATIC 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:

```
STATIC 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:

```
STATIC 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:

```
STATIC 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 class file

STATIC 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
      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!
```

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.
"%"	multiplies 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 Beginner's Guide to Gambas

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 it's 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:

```
STATIC 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 – Using Advanced Controls

In this chapter, we are going to study a few of the more advanced controls, such as the IconView, ListView, GridView, ColumnView and Tabstrip controls and 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 review the Explorer program written by Gambas creator Benoît Minisini and go through each and 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, start Gambas and select the Explorer project from the Miscellaneous section of the Example programs:

Figure 54- Choosing the Explorer example.

Once you select the Explorer project, the IDE will open up and we will need to get to the code. You can 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
```

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 '$' character for clarity. The *$sPath* variable is declared as PRIVATE and used to hold the name of the current file path.

```
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.

```
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 routine, 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, 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)
```

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 var 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 hPictParDir AS Picture ' added by the author
  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
```

The next statement is either not documented in Gambas or the documentation is unreachable on the Gambas website. However, this call appears to increment the Busy property value that is defined in the Application class. It

seems to be used as a semaphore so other processes can test the flag before trying to get computer time (remember, most applications use some sort of OS-implemented round-robin CPU scheduling routine to share processor resources). Here, it is basically a flag that tells all the other calling processes that they have to wait until until this process is not busy anymore (indicated by decrementing the value of the Busy property). This seems to be how Gambas sets flags to prevent critical processes from being interrupted.

```
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$**[14] function to convert the system charset *(the set of characters the operating system has provided as default)* to what the user has defined as the desktop charset *(the set of characters the user has chosen to see from their desktop)*. Note that **Conv** is a synonym for **Conv$** and you can use either one interchangably.

```
ME.Title = Conv($sPath, System.Charset, Desktop.Charset)
```

Now, 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["folder.png"]
hPictParDir = Picture["ParentFolder.png"]'added by the author
hPictFile = Picture["file.png"]
```

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 the 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 control, and the final

---

14 Common string functions do not deal with UTF-8 very well. In UTF-8, a character can have 1-3 bytes. The Gambas String class methods are able to handle UTF-8 so they should be used whenever possible.

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 statement will be executed in the code below:

```
IF NOT $bHidden THEN
```

This next line of code is a little tricker to decipher. The **Stat** function takes a filename string as its parameter. In this case, the current path held in global variable *$sPath* is catenated with the work string *sFile* using the **&/** symbol combination. This is a special catenation symbol used in Gambas specifically to catenate filenames. The **Stat** function is called with the catenated 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 'IF Stat
  ENDIF 'IF NOT $bHidden
```

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) 'we add it to the directories key
  ELSE 'not a directory and we add it to the "F" key as a file
    cDir.Add("F" & sFile)
  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
```

We also want to set the IconView **.Edit** property to TRUE so the user can rename items:

```
    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
  '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 MessageBox with the error text displayed:

```
Message.Error(Error.Text)
```

```
END 'of our RefreshExplorer routine
```

This subroutine is called if the FExplorer form is resized generating a resize event. It will move our IconView control to the top left corner and adjust width and height to accommodate the new size of the window:

```
PUBLIC SUB Form_Resize()
  ivwExplorer.Move(0, 0, ME.ClientW, ME.ClientH)
END
```

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, 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, we declare a new form to show the destination contents, hForm in a separate window if the control key is pressed when the destination is

selected.

```
DIM sNewPath AS String
DIM hForm AS Form
```

This if check 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
```

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 MessageBox crediting the code to 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 oversight by simply 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 Gambas language syntax 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 graphical user interface type project. When the IDE appears after you go through the project wizard, create simple form that looks like this:

A Beginner's Guide to Gambas
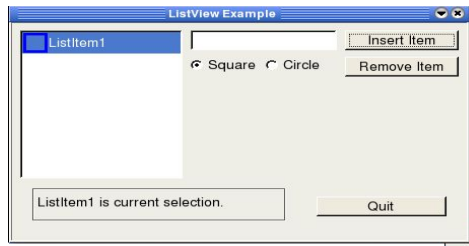
Figure 55- Layout for our ListView example.

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 botton of the form is a TextLabel named TextLabel1 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
  DIM picCircle AS NEW Picture

  picCircle.Load("circle.png")
  picSquare.Load ("square.png")
  'This will add an item to the ListView with a starting entry
  ListView1.Add ("ListItem1","ListItem1", 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.MoveCurrent
  ListView1.Item.Selected = TRUE
  TextLabel1.Text = ListView1.Item.Text & sStatus
END
```

159

In the ListView1_Click subroutine, the first line of code moves the internal cursor to the most current item and highlights it by setting the Item.Selected property to TRUE in the second line. 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.Load("square.png")
  ELSE
    picToUse.Load("circle.png")
  END IF
```

Get the current item (or do nothing) if the ListView is empty:

```
  ListView1.MoveCurrent()
```

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

    ListView1.Item.Selected = TRUE 'Selects the current item

    sStatus = " selected."

    ListView1_Click 'this will force an update

  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
```

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 Gambas Icon Edit Tool

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 Gambas 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 subMenu item. You will see this dialog pop up:
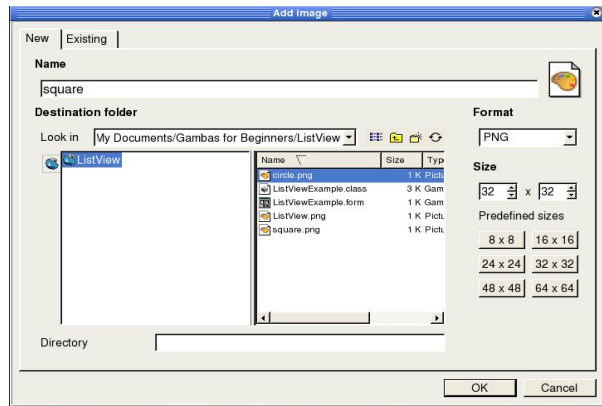
Figure 56- Creating a new Icon image in Gambas.

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 Icon Editor ToolBox and create a blue square and save it with the disk-shaped icon.
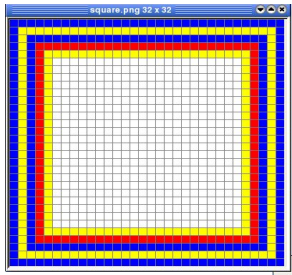
Figure 57- Our square icon image.
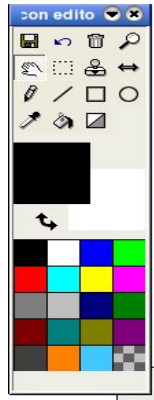
Figure 58- Icon Editor ToolBox .
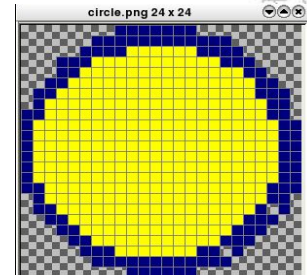
Figure 59- Our circle icon image.

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 Gambas language syntax 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 another example program from the Gambas IDE. Start Gambas and select the Examples | Basic | TreeView sample program. When the IDE opens up, you should see something similar to Figure 60 on the following page.
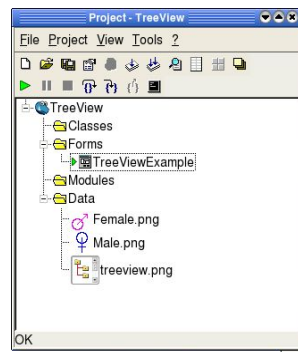
Figure 60- The TreeView Project window.

Next, double-click on the TreeViewExample 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: I'll keep the entries text and its key 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 item Frank will be a child of Sally.

```
  TreeView1.Add ("Bill","Bill", picMale)
  TreeView1.Add ("Ted","Ted",picMale,"Bill")
  TreeView1.Add ("Sally","Sally",picFemale,"Bill")
  TreeView1.Add ("Frank","Frank",picMale,"Sally")
```

After adding the items, we will move the cursor to the last item added using the TreeView1.MoveCurrent method and highlight it by setting the TreeView1.Item.Selected property to TRUE. Finally, we set the TreeView1.Item.Expanded property to TRUE to allow the item to collapse or expand when the user clicks on the plus/minus icon. *(NOTE: the Gambas Wiki documentation for this property lists it as Expand, not Expanded. Using Expand will **NOT** work.)*

```
  TreeView1.MoveCurrent
  TreeView1.Item.Selected = TRUE
```

```
    TreeView1.Item.Expanded = TRUE
END
```

If the user clicks on an item in the TreeView, we want to record the Click event in our event stack and update the stack display (TextArea1.Text).   We then increment the event number by one.   Note that in updating the TextArea1.Text value, we simply take the new event and append everything else in the TextArea1.Text memory after the linefeed character, Chr(10). The net effect of this call in an insertion of the new event at the top of the displayed text.

```
PUBLIC SUB TreeView1_Click()
  'This just updates our event stack
  TextArea1.Text = "Event(" & intEventNumber & "): Click" & Chr(10) &
TextArea1.Text
  intEventNumber = intEventNumber + 1
```

Next, 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:

```
  'This little check just updates our label so that we know how many
  'children an entry has.
  IF TreeView1.item.Children > 1 THEN
    textlabel1.Text = (TreeView1.Item.Text & " has " &
TreeView1.Item.Children & " children.")
  ELSE IF TreeView1.item.Children = 0 THEN
    textlabel1.Text = (TreeView1.Item.Text & " has no children.")
  ELSE
    textlabel1.Text = (TreeView1.Item.Text & " has 1 child.")
  END IF
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 variables. The picToUse 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 picToUse AS NEW Picture
```

```
  DIM sParent AS String

  IF Textbox1.Text <> NULL THEN
    IF RadioButton1.Value THEN
      picToUse.Load("Male.png")
    ELSE
      picToUse.Load("Female.png")
    END IF
    'Gets the parent item: the current item, or nothing is the TreeView
    'is void

    IF NOT TreeView1.MoveCurrent() THEN
      sParent = TreeView1.Key
    ENDIF
```

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,picToUse, 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,picToUse, sParent)
    TextBox1.Text = "" 'This empties out textbox
```

This next line will update our label and reflect the new number of kids:

```
    TreeView1_Click
```

This 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.

```
    TreeView1.Item.EnsureVisible
  END IF
END
```

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 Remove method in the second line, below:

```
IF TreeView1.MoveCurrent() THEN RETURN
'Lets remove the current cursor item

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
  TreeView1_Click
  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
  TextArea1.Text = "Event(" & intEventNumber & "): Collapse" & Chr(10)
& TextArea1.Text
  intEventNumber = intEventNumber + 1
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.

```
PUBLIC SUB TreeView1_DblClick()
  'This just updates our event stack
  TextArea1.Text = "Event(" & intEventNumber & "): Double Click" & Chr
(10) & TextArea1.Text
  intEventNumber = intEventNumber + 1
END


PUBLIC SUB TreeView1_Select()
  'This just updates our event stack
  TextArea1.Text = "Event(" & intEventNumber & "): Select" & Chr(10) &
TextArea1.Text
  intEventNumber = intEventNumber + 1
END
```

This next routine is dead code. It never gets executed because the Button2_Click subroutine takes care of the removal of an item. To use it effectively, it should be called from the Button2_Click subroutine just before the line calling the TreeView1_Click subroutine. If this is done, then the event stack would be properly updated, first with the delete and then the click events.

```
PUBLIC SUB TreeView1_Delete()
  'This just updates our event stack
  TextArea1.Text = "Event(" & intEventNumber & "): Delete" & Chr(10) &
TextArea1.Text
  intEventNumber = intEventNumber + 1
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, as described previously, and increments the event counter, intEventNumber, by one.

```
PUBLIC SUB TreeView1_Expand()
  'This just updates our event stack
  TextArea1.Text = "Event(" & intEventNumber & "): Expand" & Chr(10) &
TextArea1.Text
  intEventNumber = intEventNumber + 1
END


PUBLIC SUB Button3_Click()
```

```
  TextArea1.Text = ""
  'IntEventNumber = 0
END
```

The Help | About menu item generates a click event that calls this next subroutine. It simply gives credit to the author of this example, C. Packard.

```
PUBLIC SUB About_Click()
  Message.Info ("TreeView example written by C. Packard." & Chr(10) &
"Aug 2004")
END
```

If the user double-clicks on an item in the TreeView control, it sets off a Activate event. It is basically the same as a DblClick 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_Activate()
  'This just updates our event stack
  TextArea1.Text = "Event(" & intEventNumber & "): Activate" & Chr(10)
& TextArea1.Text
  intEventNumber = intEventNumber + 1
END
```

This next routine is also dead code. It never gets executed because the 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
  TextArea1.Text = "Event(" & intEventNumber & "): Rename" & Chr(10) &
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 Gambas 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 Gambas language syntax 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:

A Beginner's Guide to Gambas
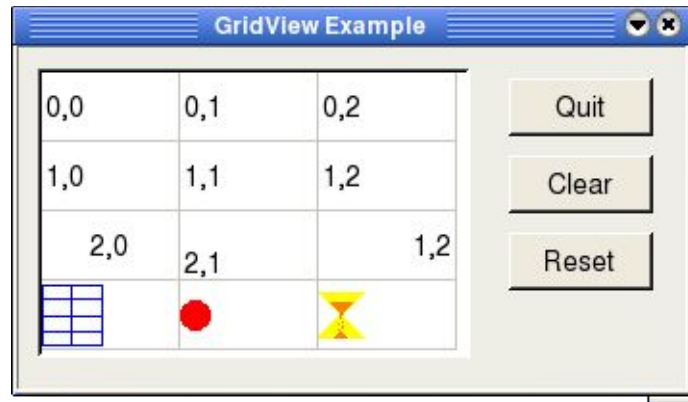
Figure 61- What our GridView will look like.

To begin, start Gambas and select the New Project option, choosing to create a Graphical User Interface project. Make the project translatable and form controls public by checking the two boxes in the Wizard. Name the project GridView Example and put it in the GridView directory. 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. Try to get them reasonably close to what you see in the picture above if you can. Create a new form that is a startup class form and put the GridView control, named GridView1 on the form as shown above. Then add the three buttons, named Button1, Button2, and Button3 to the form. Make the Button.Text properties reflect what is shown in the picture above.

Now, we have the form designed and it looks like what we expect. Lets start coding. 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()
  Form1.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.Load("GridPic.png")

hPic2 = NEW Picture
hPic2.Load("GridPic2.png")

hPic3 = NEW Picture
hPic3.Load("GridPic1.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 = 72
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 = "1,2"
```

Finally, we will add our three icons to the grid:

```
  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
```

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:
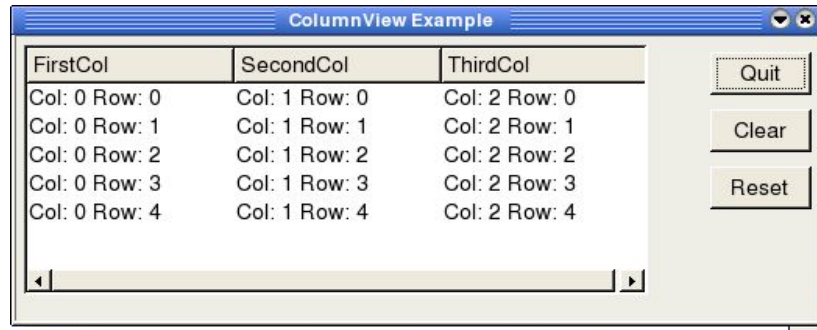
Figure 62- Our ColumnView example.

Our simple example will create three columns and 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 so that is what we will do.

```
PUBLIC SUB Button3_Click()
  _new
END
```

That is all there is to building and populating the ColumnView control. Next, we will take a look at how to use the various layout controls in Gambas.

Layout Controls – HBox, VBox, HPanel and Vpanel

All of the controls in this group are similar in that they try to arrange controls they contain automatically. They do so by using the .Arrangement property which relies upon constants that are defined in the Arrange class. Constants used by the Arrangement property of these container controls include the following: Fill, Horizontal, LeftRight, None, TopBottom, and Vertical. If you do not specify an .Arrangement property value, then the control defaults to whatever it is designed to do, i.e., HBox would arrange controls horizontally, VBox vertically, etc.

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 Gambas language syntax 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 Gambas language syntax 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 graphical user interface program. Name it Layouts and create a startup class form, Form1. For this application, we will need to use some icons. Most Linux-based distributions have icon files stored in the folder named:

<p style="text-align:center">file:/usr/share/icons/default-kde/32x32/actions</p>

Your system may be different. Regardless of where you find them, pick out seven icons and copy them to the Layout 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 I chose to use for this project:

Figure 63- 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. Here is how our form will look in design mode:
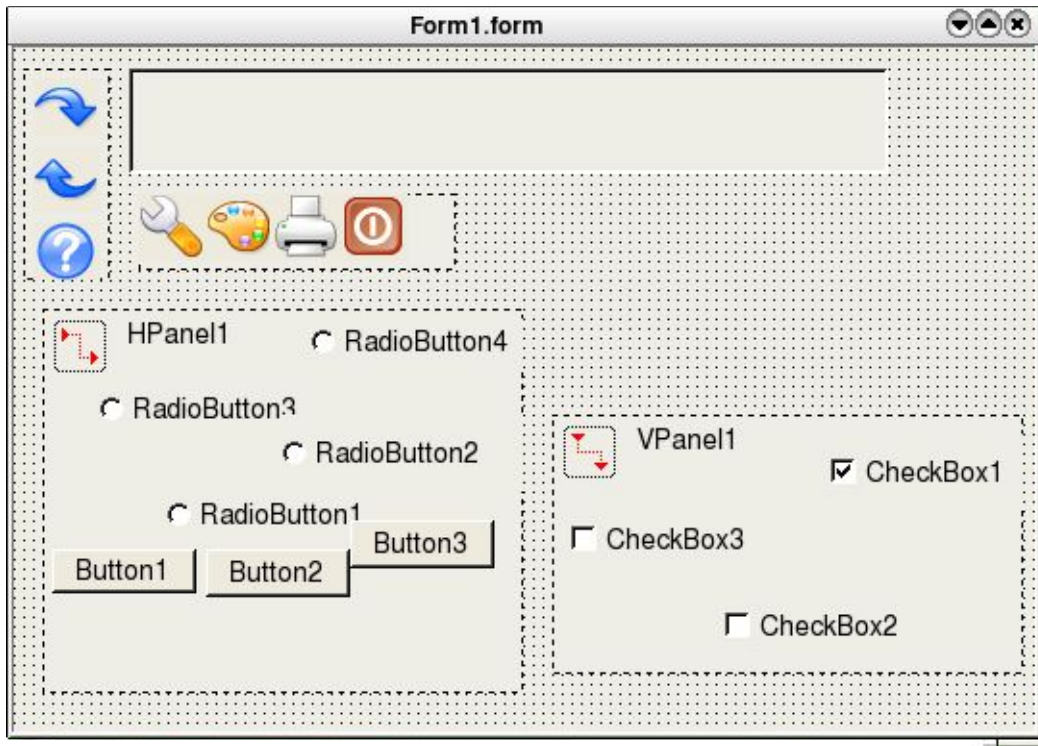
Figure 64- Form1 design mode showing layout of our controls.

From the top left of the figure above, we start with a VBox and a TextLabel. Below the TextLabel is the HBox. When you create these controls, Gambas 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. 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 = "RadioBtn 1 clicked."
END


PUBLIC SUB RadioButton2_Click()
  TextLabel1.Text = "RadioBtn 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
```

Now save your work and execute this simple program. You should see something similar to this:

Figure 65- Layout program when it starts up.

Notice how the RadioButtons in the HPanel have aligned vertically from RadioButton4 to RadioButton1 and, because 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 Gambas 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 Gambas language syntax 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 66- 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 that 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 graphical user interface program. Name the project Tabs, click through the Project Creation Wizard and when you get to the Gambas IDE, make a new form, Form1 that is a startup class. 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 a TextLabel and a button to our form. The TextLabel 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. Here is what your form should look like as you start to design the interface:

Figure 67- Tab Project Form1.form Design.

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. Gambas is design 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. For Tab0 we will add a frame and place two RadioButtons in it, like this:
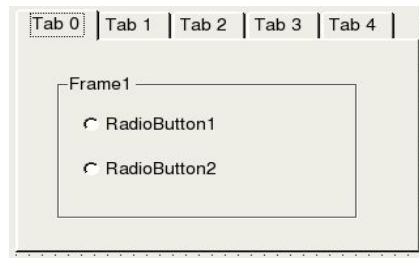
Figure 68- Tab0 layout.

Next, double-click on each RadioButton and enter this code:

```
PUBLIC SUB RadioButton1_Click()
 TextLabel1.Text = "You have clicked:<br><center><strong>RadioButton1"
END

PUBLIC SUB RadioButton2_Click()
 TextLabel1.Text = "You have clicked:<br><center><strong>RadioButton2"
END
```

Here is what we will do for Tab1:

Figure 69- 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 into 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*). In order for the Panel control to accept the ToolButtons as children, you must individually click the ToolButton on the ToolBox and create each one just as you did with 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 70- 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 clicked:<br><center>the <strong>Configure Icon"
END
```

```
PUBLIC SUB ToolButton2_Click()
 TextLabel1.Text = "You have clicked:<br><center>the <strong>Help Icon"
END

PUBLIC SUB ToolButton3_Click()
 TextLabel1.Text = "You have clicked:<br><center>the <strong>Exit Icon"
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:
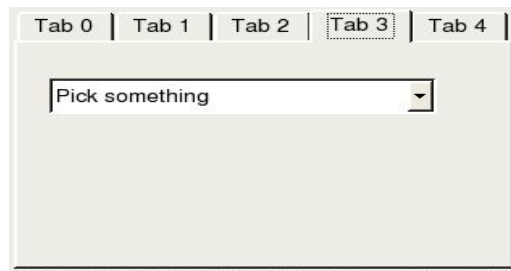
Figure 71- Tab3 layout with a ComboBox.

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:
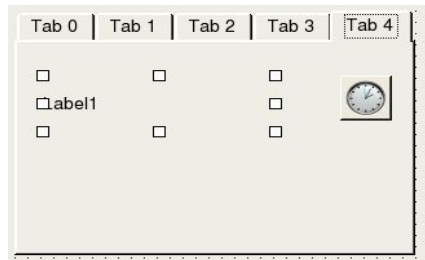
In order for the timer to work, we must enable it when the form first starts. Double-click on the form somewhere except on a control and you will see the Form_Open() subroutine appear in the code window. Add this code to the routine:

```
PUBLIC SUB Form_Open()
 Form1.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. Run the program and see how TabStrips work. Next, we move to File operations.

Chapter 9 – Working with Files

We are going to introduce you to Gambas file management and input/output (*file i/o*) operations in this chapter. Gambas has a full-featured set of functions and subroutines to support nearly any kind of file i/o operation you may need to implement in your programs.

- ✔ Access
- ✔ Dir
- ✔ Eof
- ✔ Exist
- ✔ IsDir / Dir?
- ✔ 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 Gambas language syntax 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/rabbit", gb.Write OR gb.Exec)
```

would return

True

while the statement

```
PRINT Access("/root", gb.Write)
```

will return

False

Dir

The Dir function returns a string array that contains the names of files located in *Directory* that matches 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 kind of character.
?	Any single character of any type.
[abc]	Any character specified between the brackets.
[x-y]	Any character that lies within the specified interval.
[^x-y]	Any character that is not found within the interval.

The special generic character \ prevents the character following it from being interpreted as a generic. Standard Gambas language syntax 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 Gambas language syntax 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 Exist function returns TRUE if a file or a directory exists. Using ~ as directory identifier does not work. If the path specified does not exist, FALSE is returned. Gambas language syntax is:

```
Boolean = Exist ( Path )
```

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 / Dir?

The IsDir function returns TRUE if a path points to a directory. If the path does not exist or is not a directory, this function returns FALSE.   Gambas language syntax for IsDir is:

```
Boolean = IsDir ( Path )
```

Some examples of how to use IsDir that were provided on the Gambas documentation Wiki are:

```
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 .LastUpdate
.Mode       .Perm        .SetGID     .SetUID     .Size
.Sticky     .Time        .Type       .User
```

The standard Gambas language syntax 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
```

Temp / Temp$

The Temp$ function returns a unique file name useful to create temporary files. The filename created will be located in the */tmp* directory. Gambas language syntax is:

```
File name = Temp$( )
```

Example:

```
PRINT Temp$()
/tmp/gambas.0/12555.1.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. Gambas language syntax for OPEN is:

```
OPEN File name FOR [ READ ] [ WRITE ] [ CREATE | APPEND ] [ DIRECT ]
[ WATCH ] [ BIG | LITTLE ] AS # Variable
```

If the APPEND keyword is specified, then the file pointer is moved to the end of file just after the file is opened. If the DIRECT keyword is specified, 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, the event handler File_Read() is called.
- If at least one byte can be written to the file, the event handler File_Write() is called.

If the BIG or LITTLE keyword is specified, all subsequent READ and WRITE operations on this file will use big-Endian or little-Endian data representation. *Variable* receives the object that represents the opened stream. 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
    OPEN sCurrentFileName FOR READ AS hFileIn
    sCurrentFileLength = Lof(hFileIn)
    WHILE NOT Eof(hFileIn)
       LINE INPUT #hFileIn, sInputLine 'LINE INPUT is 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 Gambas language syntax 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

   OPEN "The Raven.txt" FOR READ AS #hFile
   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 the LINE INPUT call to read data from a binary file because it will lose the linefeed characters. For reading data from binary files, you must use the READ command instead:

READ #hFile, OneLine, -256

where **-256** is the maximum number of bytes you want read (preceded with a dash) symbol. The -256 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 Gambas language syntax for READ is:

READ Variable [, Length]

If READ is used with file operations, the Gambas language syntax 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 Gambas

language syntax for standard input streams:

```
WRITE Expression [ , Length ]
```

and for writing to a file, this is the convention:

```
WRITE # File , Expression [ , Length ]
```

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 Gambas language syntax 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 Gambas language syntax 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 Gambas language syntax 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 System.Home &/ ".gambas/gambas.conf" TO"/mnt/save/gambas.conf.save"
```

The KILL function removes a file. The Gambas language syntax is **KILL Path** and whatever exists in Path is permanently deleted. RENAME simply changes the name of a file or folder from *Old name* to *New name*. Here is an example:

```
RENAME Oldname AS Newname
```

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 Gambas language syntax is:

```
MKDIR "file:/rittingj/My Documents/Gambas/test"
RMDIR "file:/rittingj/My 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 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:
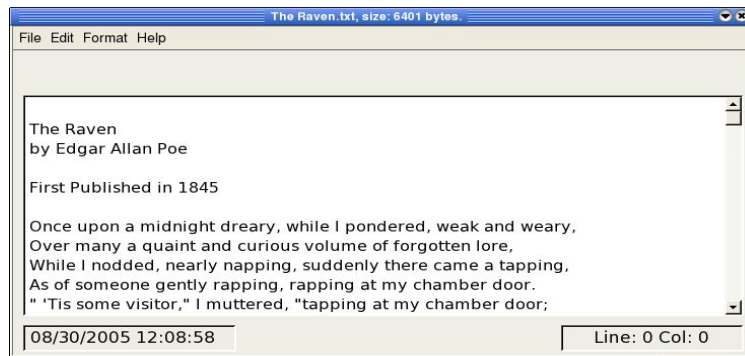
Figure 72- The FileOps program at runtime.

In this program, we will create a main menu, a text area to display and edit text with, and we will create 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 Form1.form, will look like this in design mode:
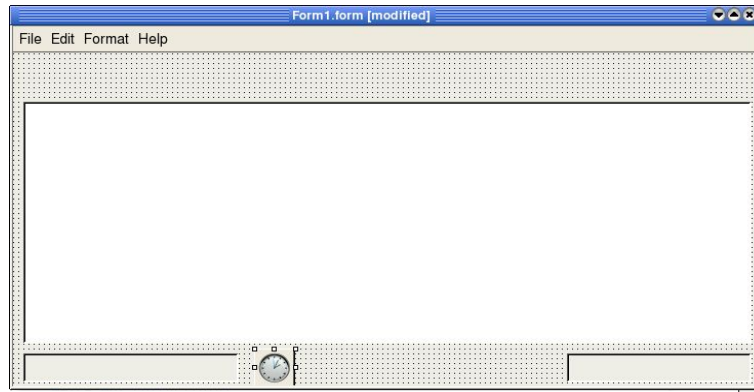
To start, load Gambas and create a new graphical user interface project. Name it FileOps and when you get to the Gambas IDE, create a new form, Form1 and make is a startup class. When the form displays on your screen, we are going to begin by creating the menus. You can press **CTRL–E** to bring up the Menu Editor. Add the following menu items shown in the tables below using the editor. For the icons, you will need to copy icons from the same directory we specified in the last chapter. Most Linux-based distributions have icon files that are usually stored in the folders namedfile:/usr/share/icons/default-kde/32x32/actions orfile:/usr/share/icons/default-kde/32x32/applications. Your particular system may be slightly different. Regardless of where you find the icons, pick out icons appropriate for our program and copy them to the FileOps folder. Here are the 17 icons I chose to use for this project:

File New   *File Open*   *File Save*   *File SaveAs*   *FilePrint*   *Exit*

Edit SelectAll   *Edit Undo*   *Edit Redo*   *Edit Cut*   *Edit Paste*   *Edit Copy*

Edit Configure   *Format Colors*   *Format Fonts*   *Help About*   *Help Contents*
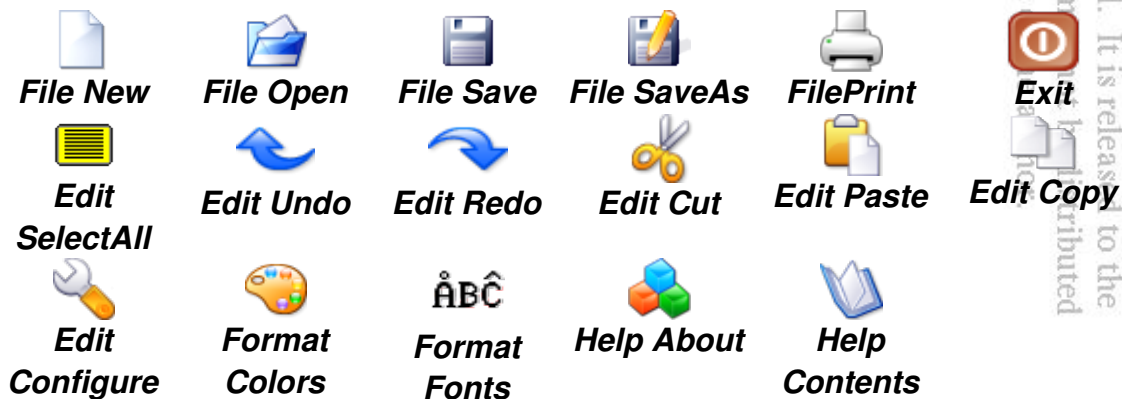
A Beginner's Guide to Gambas

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 was created using the Gambas Icon Editor.

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"]
FilePrintItem	"Print"	Picture["fileprint.png"]
FileExitItem Menu	"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)
FomatColorItem	"Colors"	Picture["colorize.png"]
FormatFontItem	"Font"	Picture["charset.png"]

HelpMenu

HelpContentsItem	"Contents"	Picture["contents.png"]
HelpAboutItem	"About"	Picture["buildingblocks.png"]

Here is the code for we will use for Form1. 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 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
    OPEN sCurrentFileName FOR READ AS hFileIn
    sCurrentFileLength = Lof(hFileIn) 'to show how to get file size
    Form1.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 Form1.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 (*.*)"]
```

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 Form1.Caption property to the name of the newly opened file and clear the current TextArea1 display by invoking the Clear method.

```
    sCurrentFileName = Dialog.Path
    Form1.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 what is 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 MessageBox 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 the 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 catenate 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 cancelled 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 MessageBox 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)
    Form1.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:

```
PUBLIC SUB FilePrintItem_Click()

  'integer variables to hold our margin values
  DIM iLeftMargin AS Integer
  DIM iTopMargin AS Integer
  DIM iRightMargin AS Integer
  DIM iBottomMargin AS Integer
```

```
'string variables we will use
DIM arsTxtToPrint AS String[]
DIM iTxtPosY AS Integer
DIM sTxtLine AS String
DIM strMsg AS String

'integer variables our program uses to calculate page dimensions
DIM iPrinterAreaWidth AS Integer
DIM iPrinterAreaHeight AS Integer
```

We **MUST** call the Printer.Setup() subroutine before we actually try to print something. It is *mandatory* to call this function in Gambas.

```
IF Printer.Setup() THEN RETURN
```

The next call ensures the default paper size is set to 8.5 by 11 (Letter). If you are not in the United States, it should probably be set to "A4" as that is the standard used in most places outside the U.S. The Gambas Printer.Setup dialog defaults to "A4" settings every time you call it, so if you are using a printer that only uses Letter size paper, over-riding this property is the easiest way to handle that situation.

```
'over-ride the printer.setup dialog so you don't have
'to set this every single time you print something
Printer.Size = "Letter" 'or A4, or whatever you use

'older printers do not seem to be set to DPIs lower than 600 in GB
'so we will set it manually for an older HP Laserjet III printer
'if your printer is newer, you can comment this line out
Printer.Resolution = 300 'my HPLJIII settings
```

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 top 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
```

The calculation for obtaining *iPrinterAreaWidth* and *iPrinterAreaHeight* subtract the left/right and top/bottom margins from the total page width/height. For Letter sized paper (8.5 x 11) the width value would be 8.5 inches times 300 dpi or 2550 and the height would be 11 inches times 300 dpi or 3300. Our calculation subtracts 600 (300 for left margin + 300 for right margin) from both the width and the height values returned from the Printer.Setup call. Next, we need to build the output string for our Message.Info call and display that data to user:

```
'build a string for MessageBox with data from printer setup call
  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 Gambas 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:

```
'use an array of strings to store each line of text in TextArea1.Text
  arsTxtToPrint = Split(TextArea1.Text, "\n")
```

We are ready to print. To do so, you **MUST** first 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 for Printer
```

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,iRightMargin,iBottomMargin)

'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
   iTxtPosY  = iTxtPosY + Draw.TextHeight(sTxtLine)

   '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,iRightMargin,iBottomMargin)
   ENDIF 'our test to see if line falls past bottom margin
```

Now, we will 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 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 we calculated above) and omitted the bottom margin since it is maintained in the code of the FOR LOOP above. 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,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 example 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 Form1.form and press **CTRL-E** to bring up the menu editor. Below the File | Print menu item, add a new item, *FileSysPrintItem* and give it the caption Print (system method). Create a click event and add this code:

```
PUBLIC SUB FileSysPrintItem_Click()
```

```
  DIM aString AS String

  aString = "pr -h \" " & sCurrentFileName & "\"" & "  -o 10 <  \"" &
sCurrentFileName & "\"  | lpr "

  message.Info(aString,"Ok")
  SHELL aString
END
```

It looks a lot more difficult that it really is. Let's go line by line through the code. First, we declare a string variable, *aString*. Next we need to build the string that will be passed off to the system as a console command and assign it to your *aString* variable. 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, in our program, is global in scope. This filename has to be embedded in quote marks so we need to use the \ character before the opening quote mark and terminate this part of the string with another quote mark. The next part of the string:

```
& sCurrentFileName & "\""
```

concatenates the filename and the terminating quote for the filename passed in as a string variable, once again preceded by a \ character. The next quote mark terminates this portion of the string. The next part of the command to concatenate is:

```
& " -o 10 <  \""
```

which adds the -o parameter (which specifies an offset) and a value of 10 spaces. The < character is used on the command line to designate the input stream comes from the filename that follows the < character. In this case, we need to delimit our string with quotes again, so the \ character is used to allow the quote character to be part of the output string. The next quote character begins the quoted filename string. The last part of our command is:

```
sCurrentFileName & "\"  | lpr"
```

It specifies the file to read (and print) and terminates the filename with quote marks as explained before. The final part of the string is the | character which *pipes* (redirects) the output to the line printer (lpr). In all, it takes one variable declaration and three lines of code to print a file using this technique. Either technique will work so choose the one that suits you best. Another method that you can use if you have KDE installed is to print to a postscript file and send the .ps file to '*kprinter*'. This will provide you with access to the KDE print dialogs and all of the CUPS printers and features installed on your system.

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 0
         File.Save(sCurrentFileName, TextArea1.Text)
      CASE 1
      CASE 2
      CASE ELSE
 END SELECT
 ME.close
END
```

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 be 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.Selection(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.Selection(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 Gambas 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
```

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 Form2.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()
```

```
 Form2.Show
END
```

Here is what Form2.form will look like:
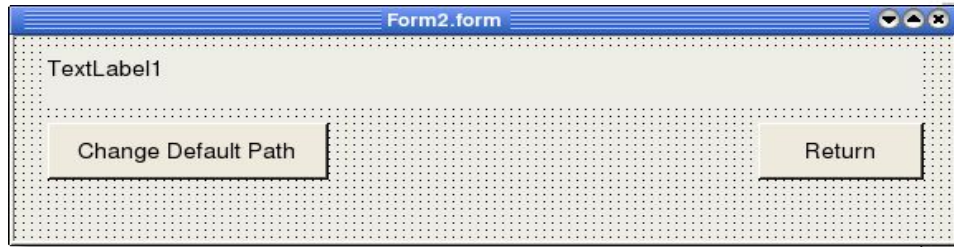
Figure 73- Form2.form design mode.

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, Form1.form in this case. When the form is opened, we will display the caption "Preferences" at the top of the window. The constructor routine _new() is called automatically. Here is the code for our Form2:

```
' Gambas class file
PUBLIC SUB Form_Open()
  Form2.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 MyPath AS String

 MyPath = application.Path &/ "FileOps.conf"
  WritePrefs(MyPath)
  Form2.Close
END
```

As you can see from above, a local string variable MyPath is declared and populated by concatenating the current application path with the configuration filename FileOps.conf. Next, we call our WritePrefs subroutine and pass the value of MyPath as a parameter. That path will be stored in the file and can be read by any other class that needs to find out this information. 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
    OPEN filename FOR READ AS #hFileIn

    '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
     ' 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 there is already a config file, open it for write
 ' if not, open for create and write
 IF Exist(MyPath) THEN
     OPEN MyPath FOR WRITE AS # hFileOut
 ELSE
     OPEN MyPath FOR WRITE CREATE AS # hFileOut
 ENDIF

 ' print our data to the file
 PRINT # hFileOut, MyPath

 ' close the file
 CLOSE # hFileOut
END
```

As you can see, it is pretty simple but it 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 Gambas standard 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
```

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()
 SHELL "konqueror \"file:/root/My Documents/Gambas for Beginners/FileOps/Help.html\""
END
```

The shell call above points to a file named Help.html. For purposes of

demonstrating this feature, simply copy any .html file on your system to the program directory and rename it help.html. As an alternative, you can create your own .html file if you know how to do that. That is what *real* programmers would do. ☺ Here is the last thing we need to do:

```
PUBLIC SUB HelpAboutItem_Click()
 Message.Info("<b>Simple Text Editor</b> by <br>J. Rittinghouse", "OK")
END
```

Run the program and when you are satisfied, let's move on to do some math.

Chapter 10 – Math Operations

Gambas performs mathematical operations with a library of intrinsic and derived functions. According to the online encyclopedia Wikipedia[15], "*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. There is a table provided at the end of this chapter that shows you the official "developer" version of the hierarchy of operations, as defined in the **gb_reserved_temp.h** Gambas source file.

---

15 See URL: http://en.wikipedia.org/wiki/Intrinsic_function.

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 Gambas language syntax is:

```
Abs(number)
```

The following example uses the Abs function to compute the absolute value of a number. Try this on your console:

```
STATIC PUBLIC SUB Main()
  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. $Y = ACos(X)$ returns the arc-cosine for each element of X. For real elements of X in the domain[-1,1], $ACos(X)$ is real and in the range . For real elements of X outside the domain [-1,1], $ACos(X)$ 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 $X >= 1$. The range of the hyperbolic arc-cosine function is limited to non-negative real numbers *Y* such that $Y >= 0$. Gambas language syntax is as follows:

```
value = Acsh ( Number )
value = ACosh ( Number )
```

Here is an example to try on the console:

```
STATIC 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 used for x is not in the range [-1,1]. Standard Gambas language syntax is:

```
value = Asn ( Number )
value = ASin ( Number )
```

Here is an example:

```
STATIC 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 expressed in radians. Standard Gambas language syntax is:

```
value = Asnh ( Number )
value = ASinh ( Number )
```

An example is as follows:

```
STATIC 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 Gambas language syntax is:

```
value = Atn ( Number )
value = ATan ( Number )
```

Example:

```
STATIC 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 Gambas language syntax 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():

```
STATIC 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() in 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 π/180. To convert radians to degrees, multiply radians by 180/π . Standard Gambas language syntax 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:

```
STATIC PUBLIC SUB Main()
 DIM MyAngle AS Float
 DIM MySecant AS Float

 MyAngle = 1.3                    ' Define angle in radians.
 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 Gambas language syntax is:

```
Value = Cosh ( Number )
```

Here is an example to try on the console:

```
STATIC 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 Gambas language syntax for Deg and Rad is:

```
Value = Deg ( Angle )
Value = Rad ( Angle )
```

An example to try on the console is:

```
STATIC PUBLIC SUB Main()
  PRINT Deg(Pi / 2)
  PRINT Rad(90)
  PRINT Rad(180) – Pi
END
```

The console responds with:

```
90
1.570796326795
0
```

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 Gambas language syntax is:

```
Value = Exp ( Number )
```

Here is an example:

```
STATIC 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:

```
STATIC 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 -π. The standard Gambas language syntax is:

```
Value = Fix ( Number )
Value = Frac ( Number )
```

Example:

```
STATIC 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 Gambas language syntax is:

```
value = Log ( Number )
```

Here is an example:

```
STATIC PUBLIC SUB Main()
 PRINT Log(2.71828)
 PRINT Log(1)
END
```

The console responds with:

```
0.999999327347
0
```

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:

$$Logn(x) = Log(x) / Log(n)$$

Here is an example to try:

```
STATIC 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 Logn(10)
```

Log10

Log10 is a derived function that computes the decimal logarithm of a number. $Log10(x) = Log(x) / Log(10)$. Gambas language syntax 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:

```
STATIC PUBLIC SUB Main()
 PRINT Max(6, 4, 7, 1, 3)
 PRINT Max(Now, CDate("01/01/1900"), Cdate("01/01/2100"))
 PRINT
 PRINT Min(6, 4, 7, 1, 3)
 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 π * *Number*. If *Number* is not specified, it is assumed to be one. Standard Gambas language syntax is:

```
Result = Pi ([Number])
```

Examples:

```
PRINT Pi
3.14159265359

PRINT Pi(0.5)
1.570796326795
```

Randomize and Rnd

Randomize initializes a pseudo-random number generator. It uses a methodology that seeds itself with the current date and time. The Gambas language syntax is:

```
Randomize (Number)
```

where the number argument can be any valid numeric expression. Randomize uses the parameter *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 Gambas language syntax 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. 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:

```
STATIC 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))
 PRINT
 Randomize

 DO WHILE Dice <> 1
  Dice = Int(Rnd(1,7))
   'produce a random number between 1 and 6 to simulate the dice throw
  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:

```
STATIC 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. Gambas language syntax is:

```
Sign = Sgn ( Number )
```

Example:

```
STATIC PUBLIC SUB Main()
 PRINT Sgn(Pi)
 PRINT Sgn(-Pi)
 PRINT Sgn(0)
END
```

The console responds with:

```
1
-1
0
```

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 θ is the vertical coordinate of the arc endpoint. As a result of this definition, the sin function is periodic with period 2π. By the Pythagorean theorem, sin θ also obeys the identity $\sin^2\theta + \cos^2\theta = 1$. The standard Gambas language syntax for Sin is:

```
Value = Sin ( Angle )
```

Example:

```
STATIC PUBLIC SUB Main()
 PRINT Sin(Pi/2)
END
```

The console responds with:

```
1
```

Sinh

Sinh is a derived function that computes the hyperbolic sine of a number. Standard Gambas language syntax for Sinh is:

```
Value = Sinh ( Number )
```

The following example uses the Exp function to calculate hyperbolic sine of an angle:

```
STATIC 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 Gambas language syntax 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:

```
STATIC PUBLIC SUB Main()

 DIM MySqr AS Float
```

```
 MySqr = Sqr(4)     'Returns 2
 PRINT MySqr & " is the result of Sqr(4)"
 MySqr = Sqr(23)     'Returns 4.795831523313
 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
```

The console responds with:

```
2 is the result of Sqr(4)
4.795831523313 is the result of Sqr(23)
0 is the result of Sqr(0)
```

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 Gambas language syntax is:

```
Value = Tan ( angle )
```

The following example uses the Tan function to return the tangent of an angle:

```
 STATIC 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
```

1

Tanh

Tanh is a derived function that computes the hyperbolic tangent of a number. The standard Gambas language syntax for Tanh is:

```
Value = Tanh ( Number )
```

Example:

```
STATIC PUBLIC SUB Main()
 PRINT Tanh(1)
END
```

The console responds with:

```
0.761594155956
```

Derived Math Functions

The non-intrinsic (derived) math functions shown in the table below are provided in Gambas from the built-in intrinsic math functions.

Gambas Derived Math Functions

Function	Derived equivalents
Inverse Sine	ASin(X) = Atn(X / Sqr(-X * X + 1))
Inverse Cosine	ACos(X) = Atn(-X / Sqr(-X * X + 1)) + 2 * Atn(1)
Hyperbolic Sine	Sinh(X) = (Exp(X) - Exp(-X)) / 2
Hyperbolic Cosine	Cosh(X) = (Exp(X) + Exp(-X)) / 2
Hyperbolic Tangent	Tanh(X) = (Exp(X) - Exp(-X)) / (Exp(X) + Exp(-X))
Inverse Hyperbolic Sine	ASinh(X) = Log(X + Sqr(X * X + 1))
Inverse Hyperbolic Cosine	ACosh(X) = Log(X + Sqr(X * X - 1))
Inverse Hyperbolic Tangent	ATanh(X) = Log((1 + X) / (1 - X)) / 2
Logarithm to base 10	Log10(X) = Log(X) / Log(N)

Below is a table listing other derived math functions that can be built using the formulas shown in the table below:

Other Derived Math Functions

Secant	Sec(X) = 1 / Cos(X)
Cosecant	Cosec(X) = 1 / Sin(X)
Cotangent	Cotan(X) = 1 / Tan(X)
Inverse Secant	Arcsec(X) = Atn(X / Sqr(X * X - 1)) + Sgn((X) -1) * (2 * Atn(1))
Inverse Cosecant	Arccosec(X) = Atn(X / Sqr(X * X - 1)) + (Sgn(X) - 1) * (2 * Atn(1))
Inverse Cotangent	Arccotan(X) = Atn(X) + 2 * Atn(1)
Hyperbolic Secant	HSec(X) = 2 / (Exp(X) + Exp(-X))
Hyperbolic Cosecant	HCosec(X) = 2 / (Exp(X) - Exp(-X))
Hyperbolic Cotangent	HCotan(X) = (Exp(X) + Exp(-X)) / (Exp(X) - Exp(-X))
Inverse Hyperbolic Secant	HArcsec(X) = Log((Sqr(-X * X + 1) + 1) / X)
Inverse Hyperbolic Cosecant	HArccosec(X) = Log((Sgn(X) * Sqr(X * X + 1) +1) / X)
Inverse Hyperbolic Cotangent	HArccotan(X) = Log((X + 1) / (X - 1)) / 2
Logarithm to base N	LogN(X) = Log(X) / Log(N)

Since Gambas does not directly provide you with built-in functions for these other 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 named MyMath. Create a new class, Class1 from the IDE and make it a startup class. Next create a new module and name it Derived. Open the new code window for the Class1.class file and enter the following:

```
' Gambas class file
STATIC 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, open the code window for the Derived.module file and enter this code:

```
' 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

'this routine currently produces an error in Gambas
'if you try to use the formula result = Log((Sqr(-x*(x+1))+1)/x)
'because the Log function only works with non-negative numbers
'by design.  This is a work-around to get you 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 * -1)*( x + 1))+1
 Temp2 = Sqr(Abs(Temp1))
 Temp3 = Log(Temp2)
 result = Temp3 * -1

 'this call will generate an error because the
 'Sqr function will not work with negative values
 'result = Log((Sqr(-x*(x+1))+1)/x)
```

```
 RETURN result
END

PUBLIC FUNCTION HArccosec(x AS Float) AS Float
 DIM result AS Float
 result = Log((Sgn(x)*Sqr(x*x+1)+1)/x)
 RETURN result
END

PUBLIC FUNCTION HArccotan(x AS Float) AS Float
 DIM result AS Float
 result = Log((x+1)/(x-1))/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 receive a prompt to enter a number >
1 or 0 to quit. Enter 4 and you should see the following results:

```
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:
```

Now, you should be able to perform almost any type of mathematical
operation necessary for most programs. You have absorbed a lot of material in

this chapter and it is time to take a break now. In the next chapter, we are going to cover the basic concepts of object-oriented (OO) programming and show you how powerful Gambas can be when you take advantage of these OO features.

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 an 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 object-oriented programming languages (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 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 Gambas 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 Gambas 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 OOPL. 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 OOPLs 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 OOPLs, 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 OOPLs 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 re-implement 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*). 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, often reflecting a common set of behaviors.

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. It is important to note that the current version of Gambas does not support the concept of *virtual dispatch*. In other words, when you invoke a method (or set/get a property value) for an object reference using the dot notation (*i.e., class.object.method),* the method that is actually executed is the method belonging to the data-type of the variable that *owns the object reference*, not the method of the real data-type of the object. In Gambas2, this issue has been corrected.

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 Form1.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 Form1.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
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]
      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 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 one parameter, the collection object we wish to store. *cTheData* is the collection passed to 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.
 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
```

Form1.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 Form1. Form1 will serve as the main interface between the user and the program and it's data. Here is what our finished form will look like:
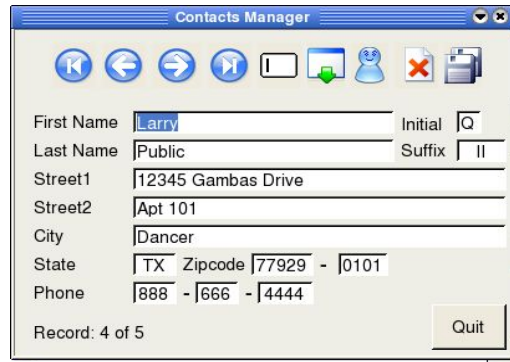
A Beginner's Guide to Gambas

Figure 74- Finished Contacts program.

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!

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 Form1 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

 'we will need a string var to use when the user searches by last name
 PUBLIC SearchKey AS String
```

Form1 Constructor

```
'the constructor routine called when form1_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"
  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 MessageBox 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 Form1.form is defined as a startup form, it is not necessary to call the Form1.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
 Form1.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 Form1.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 I used for this program are as follows:

Start of data	Prev	Next	End of data	Clear fields
Update fields	Add new	Delete current	Save to file	

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 in the figure 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 Figure 74.   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 the Labels and TextBoxes

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 you see in Figure 74 above. 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 undeclared variables when you try to run the program. At this point, your form, in design mode, should look like this:
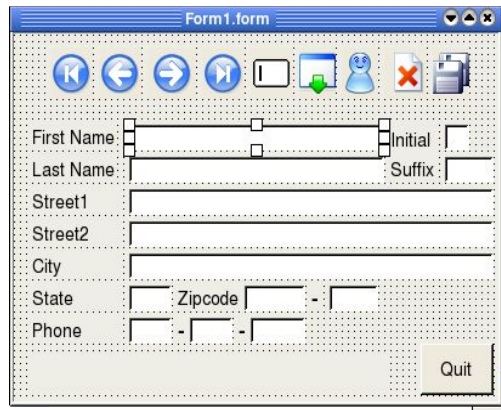
A Beginner's Guide to Gambas

Figure 75- Form1 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
   Zip5.Text = ContactRecord.Zip5
   Zip4.Text = ContactRecord.Zip4
   AreaCode.Text = ContactRecord.Areacode
```

253

```
  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
  Form1.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
```

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 (←) 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 varaible 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 (→) 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

'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 varaible just for use in this subroutine
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
          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()
  DIM sTemp AS String

  IF Len(Zip5.Text) > 5 THEN
    sTemp = Mid$( MI.Text, 1, 5)
    Zip5.Text = sTemp
  ENDIF
END

PUBLIC SUB Zip4_Change()
  DIM sTemp AS String

  IF Len(Zip4.Text) > 4 THEN
    sTemp = Mid$( MI.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 a last name to which we will use as a key for our search. Create an event for MouseDown on the form. Here is the code for the MouseDown 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:

—
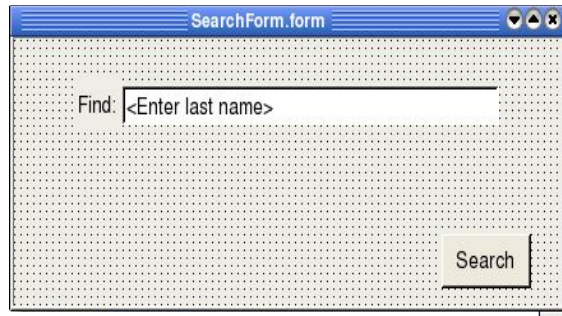
Pretty basic, just a label, a Textbox, and a button. From the IDE create a new form, named SearchForm and add the label and button as shown below. Name the button SearchBtn and the TextBox should be named SearchInput. Here is all the code that is needed for the SearchForm:

```
' Gambas SearchForm class file

PUBLIC SUB SearchBtn_Click()

  '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 Form1
  Form1.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 exactly 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 Form1 class public variable SearchKey. Also note that we fully qualified the variable name, specifying Form1.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 Form1 class file that called it.

```
' Gambas module file

PUBLIC FUNCTION SearchOn(sKey AS String) AS String
  RETURN sKey
END
```

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
```

ToolButtons again: Updating a 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 if failed
    ENDIF
    'update the form with the new data
    UpdateForm
END
```

Toolbuttons again: Deleting a 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
```

ToolButtons again: 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:

```
'/rittingj/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 used for this program, it would not work until I enclosed the full path with tick marks, as 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 */rittingj/My* (where the first space character occurs). Here is the final result, our program executing in stand-alone mode on the desktop:
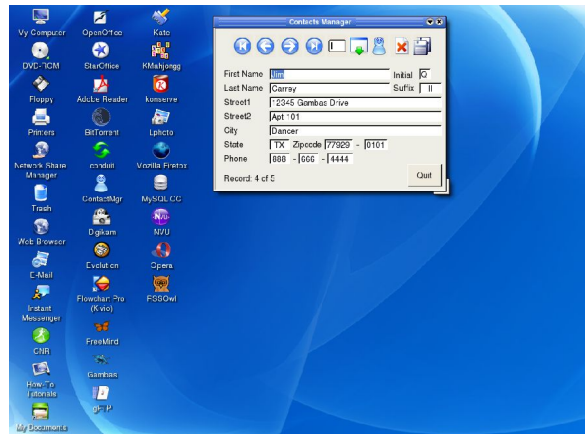

Figure 76- Contacts Manager running standalone on my desktop.

Chapter 12 – Learning to Draw

The Draw class 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.

BackColor/Background and ForeColor/Foreground

As we explained in Chapter 4, many of the properties for controls are common among all controls, so we will only explain the properties that are unique to a given control in this chapter.

```
BackColor is defined as PROPERTY BackColor AS Integer
ForeColor is defined as PROPERTY ForeColor AS Integer
```

This integer value represents the color used for the current drawing background or foreground color. It is synonymous with the Background/Foreground properties. Note that **PROPERTY** is a predefined data-type used internally in Gambas. 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 BackColor property to red, you would use this code:

```
Draw.BackColor = Color.Red
```

Alternatively, as we stated previously, 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 BackColor (*or other color-related*) property. The Color class provides two methods, RGB and HSV that you can use. The **RGB** function returns a color value from its red, green and blue components. **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.BackColor = Color.RGB(255,255,255)
```

This would set the Draw object'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

The Clip 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. The virtual class *.DrawClip* is 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 Gambas language syntax 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 Gambas language syntax:

```
STATIC SUB .DrawClip (X AS Integer, Y AS Integer, W AS Integer, H AS Integer )
```

Properties that are returned by this function include Enabled, H, Height, W, Width, X and Y.

FillColor,FillStyle,FillX,FillY

The FillColor property returns or sets the color used by drawing methods

capable of filling a polygon with a color or pattern or both. This class is static. The FillStyle property returns or sets the style used by drawing methods capable of filling a polygon with a color or pattern or both. The Fill class supports several predefined FillStyle constants used to represent fill patterns for drawing:

```
BackDiagonal      Cross         CrossDiagonal     Dense12
Dense37           Dense50       Dense6            Dense63
Dense88           Dense94       Diagonal          Horizontal
None              Solid         Vertical
```

Gambas language syntax for FillColor and FillStyle is as follows:

```
STATIC PROPERTY FillColor AS Integer
STATIC PROPERTY FillStyle AS Integer
```

The FillX property and the FillY property are used to return or set the horizontal/vertical origin of the brushes used by drawing methods capable of filling a polygon with a color or pattern or both. Gambas language syntax for FillX and FillY is:

```
STATIC PROPERTY FillX AS Integer
STATIC PROPERTY FillY AS Integer
```

Font

The Font class returns or sets the font used for rendering text on the drawing surface. Gambas language syntax 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

The Invert property is used to stipulate that all drawing primitives will combine their pixel colors with the pixel colors of the destination using an XOR operation. Gambas language syntax is:

```
STATIC PROPERTY Invert AS Boolean
```

LineStyle/LineWidth

The LineStyle property 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 Gambas language syntax is:

```
STATIC PROPERTY LineStyle AS Integer
STATIC PROPERTY LineWidth AS Integer
```

Transparent

The Transparent property indicates that some drawing methods like Draw.Text do not fill their background with anything (hence, they are transparent). Gambas language syntax 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 be creating a new graphical user interface application. Name the project gfxDemo and when the IDE appears, create a form, Form1. Make sure you specify it as a startup class and ensure the controls are public. Here is what the Form1 layout will look like at runtime:

A Beginner's Guide to Gambas

Figure 77- gfxDemo Form1 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 DrawingArea control, named *da* to the form. Place it at the top left corner of Form1 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
   da.W = form1.W — 10

   'establish height of the drawing area
   da.H = form1.H – 45

   'make the mandatory call to start draw operations
   Draw.Begin(da)
   'set a default line width of 2 pixels
```

A Beginner's Guide to Gambas

Figure 77- gfxDemo Form1 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 DrawingArea control, named *da* to the form. Place it at the top left corner of Form1 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
   da.W = form1.W — 10

   'establish height of the drawing area
   da.H = form1.H – 45

   'make the mandatory call to start draw operations
   Draw.Begin(da)
   'set a default line width of 2 pixels
```

271

```
   Draw.LineWidth = 2
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()
  Form1.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 Form1
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. The TextHeight property returns the height of a text drawing while the TextWidth property returns the width of a text drawing. Standard Gambas language syntax is:

```
STATIC FUNCTION TextHeight (Text AS String) AS Integer
STATIC 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.ForeColor = 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
    '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.ForeColor = 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

'refresh the drawing area to see what was done
da.Refresh
END ' we are done with text stuff!
```

Once you have added the code above, save your work and click the RUN button. Here is what you should see:
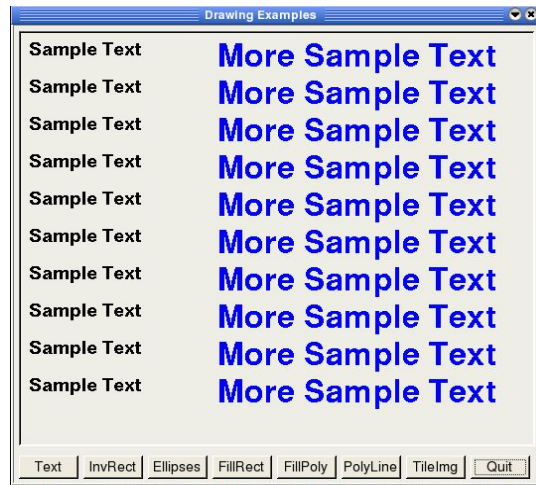
Figure 78- Results of clicking the Text Button.

Draw Primitives: Point/Rect/Ellipse/Line

The Draw.Point method draws a single pixel while the Draw.Rect method draws a rectangle. Gambas language syntax 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.ForeColor = 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)

'update the display to see what we did
da.Refresh
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:
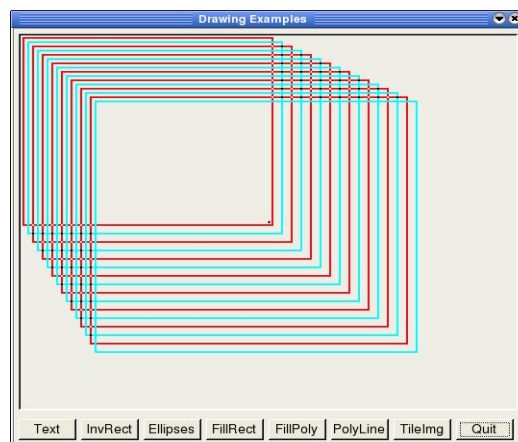
Figure 79- Results of InvRect button click. Note the tiny black crosshair center screen.

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 Gambas language syntax for this method is:

```
STATIC SUB Line(X1 AS Integer,Y1 AS Integer,X2 AS Integer,Y2 AS Integer)
```

The Draw.Ellipse method will draw an ellipse or circle. Remember, a circle is an ellipse with equal width and height parameters, which is why you do not see a Draw.Circle function in Gambas. The Gambas language syntax 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 though of as a radius along the horizontal plane, height is a radius along the vertical plane. If you specify the optional parameters of start and length, these represent starting angles (in degrees) where the drawing will start and stop. 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
   DIM y1 AS Integer
   DIM y2 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

   'clear our display before we begin
   da.Clear

   'set initial vectors
   x1 = 50
   y1 = 50
   x2 = 100
   y2 = 100
```

```
'change the color to dark blue
Draw.ForeColor = 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

   'draw a filled ellipse of 12 segments, starting at each value of i
   'and sweeping for 30 degrees, as specified in the optional parms
   Draw.Ellipse(x1, y1, x2, y2, i, 30)
NEXT

'now, turn off the fill style
draw.FillStyle = fill.None

'set our fgd color to black
draw.ForeColor = color.Black

'lets draw some lines, start x at midscreen horiz axis - 25 pixels
x1 = da.W/2-25

'start our y at midscreen on the vertical axis
y1 = da.H/2

'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

'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(x1-i*5, y1, 75, 200)

    'and moving this one right along the horiz axis
    Draw.Ellipse(x1+i*5, y1, 75, 200)

    'if we hit an even number in our loop change colors
    IF i MOD 2 = 0 THEN
       draw.ForeColor = color.Red
    ELSE
       draw.ForeColor = color.Black
    ENDIF
  NEXT 'iteration of the loop

'refresh the drawing area to see our work
da.Refresh
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:
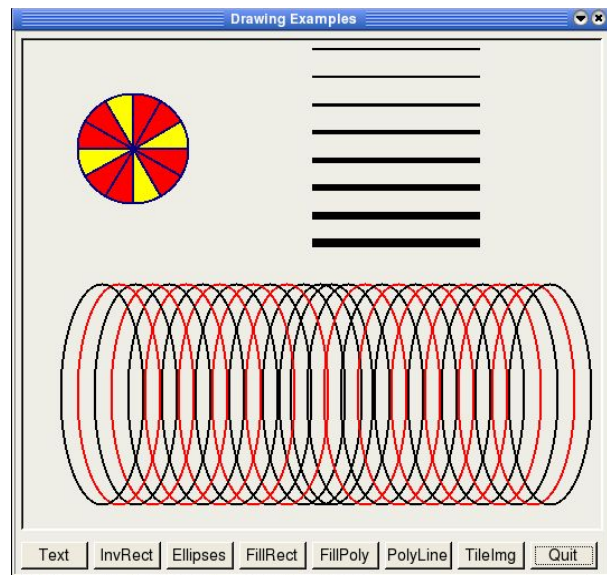
Figure 80- Ellipses demonstrates drawing lines and ellipses.

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 x1 AS Integer
   DIM x2 AS Integer
   DIM y1 AS Integer
   DIM y2 AS Integer

   'clear the drawing area
   da.Clear

   'set the rect top left corner
   x1 = 20
   y1 = 80

   'set the rect bot right corner
   x2 = x1+50
   y2 = y1+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(x1,y1,x2, y2)

   'set the rect top left corner
   x1 = 220
   y1 = 180

   'set the rect bot right corner
   x2 = x1+50
   y2 = y1+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

   'refresh to see our work
   da.Refresh
END
```

Save your work and run the program. When you click the FillRect button, you should see something similar to this:
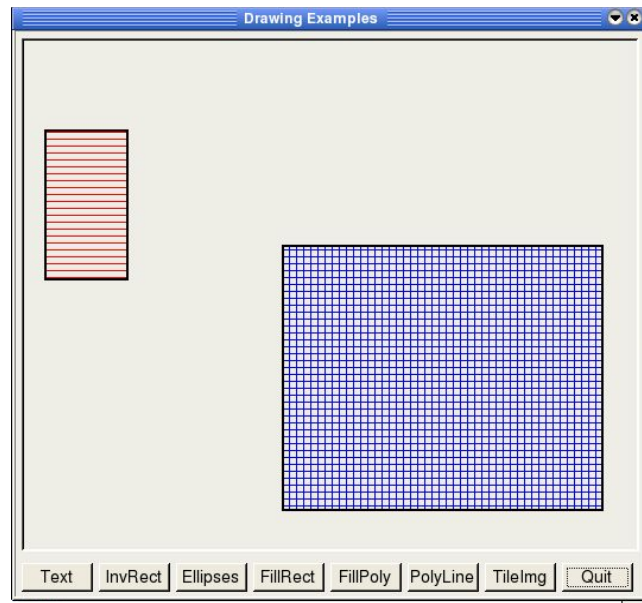

Figure 81- Output after clicking the FillRect button.

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 Gambas language syntax:

```
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 method 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 Gambas language syntax 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 Form1 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 x1 AS Integer
  DIM y1 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
  x1 = da.w/2 ' set the horiz axis to mid-screen
  y1 = 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

  'and the vert axis will be same as the second leg
  y3 = y2

  'all vertices defined, load the integer array with the triangle
```

```
    triangle = Array(x1, y1, x2, y2, x3, y3)
    '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:
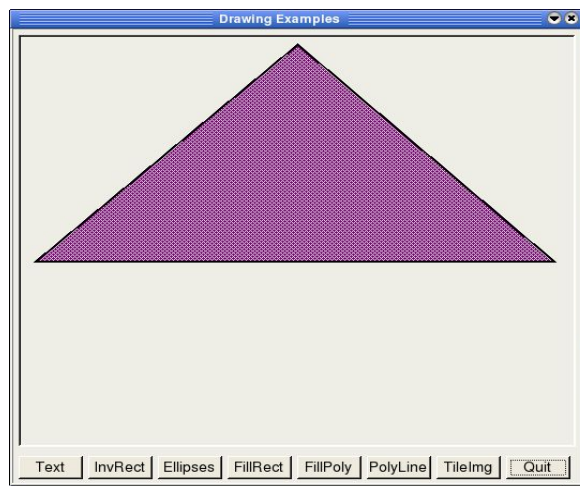
Figure 82- Using Draw.Polygon to draw a triangle.

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 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
    'uncomment the next line to try it and see
    'lines = Array(x1, y1, x2, y2, x3, y3, x4, y4, x1, y1)
    lines = Array(x1, y1, x2, y2, x3, y3, x4, y4)

    'ensure any fill styles are turned off
    draw.FillStyle = fill.None

    'set the color to dark green
    draw.ForeColor = color.DarkGreen

    'draw the lines
    draw.Polyline(lines)

    'refresh to see our work
    da.Refresh
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:
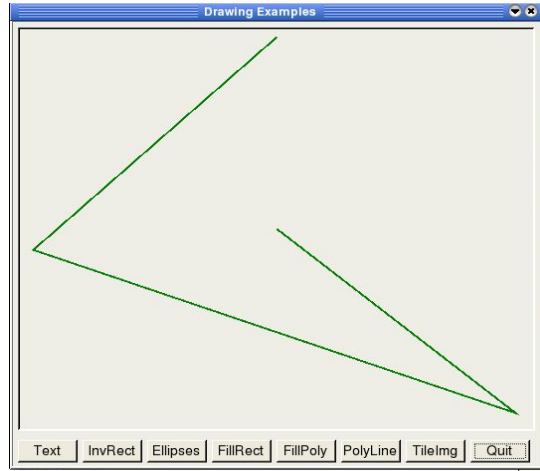

Figure 83- Using Draw.Polyline to draw lines.

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. Gambas language syntax 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. Gambas language syntax 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. Gambas language syntax 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. Gambas language syntax is as follows:

```
FUNCTION Rotate ( Angle AS Float ) AS Image
```

The function will rotate the image *n* degrees. The angle is specified in degrees, not radians.

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 Gambas language syntax 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. Even if X-Window does not manage transparency yet, each picture can have a 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. Gambas language syntax 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 a very simple example 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 graphical user interface project, named gfxDemo2. Make the project translatable with public controls and when the IDE appears, create a startup class form named Form1. From the properties window for Form1, set the resize 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

 'get the picture ojbect from the desktop using .Grab method
 p = Desktop.Grab()

 '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 fm 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
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?

Transparent is a Boolean flag that can be read or set. It indicates whether or not the picture has a mask. Gambas language syntax 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. Gambas language syntax 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. Gambas language syntax 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.   Gambas language syntax 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 Form1 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, set width, height so it is not void
  mypic = NEW Picture(170, 105,FALSE)

  'clear the drawing area
  da.Clear

  'now load the picture using the load method
  mypic.Load("Gambas Shrimp.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, j, 170,105)

  '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

  'refresh the display to see the work
  da.Refresh
END 'of the TileImg button code
```

The last thing we need to do is get an image to use for our program. For our progarm, 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 *"Gambas mascot.png"* image. 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. Now, save your work

and run the program. Click the TileImg button and you should see something similar to this:
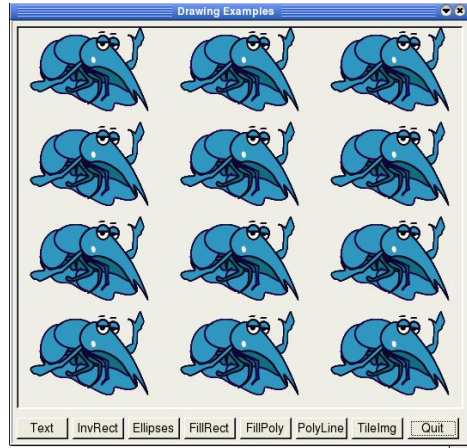
Figure 84- Using tiled images with the TileImg button of our demo program.

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 specification[16], *"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."*

Inkscape[17] 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.

In Gambas, you can load, save, clear, or copy an SVG image. The

---

16 See URL http://www.w3.org/TR/SVG/intro.html for more details on SVG.

17 See URL http://www.inkscape.org/ for more details on Inkscape.

properties that you have available to you are the most basic, namely width, height, and depth. Gambas is not designed to be an SVG editor. It simply has provisions for you to use SVG images in your program. Once you have the drawing in your program, it is up to you to do something with it. As an example, we will create a new program to simply load an SVG image and display it. First, we will need to find an SVG image. I have downloaded a freely available public domain licensed image from the Internet at http://openclipart.org to use for our demo. You can choose any image you wish for this demo, so long as it is an SVG formatted file. Now, we will create a new graphical user interface project in Gambas named gfxDemo3. Make the project controls public and translatable. Next, create a new startup class form, named Form1. We are going to add several controls: Scrollview, DrawingArea, HBox, and two buttons. Your form should look like this in design mode:
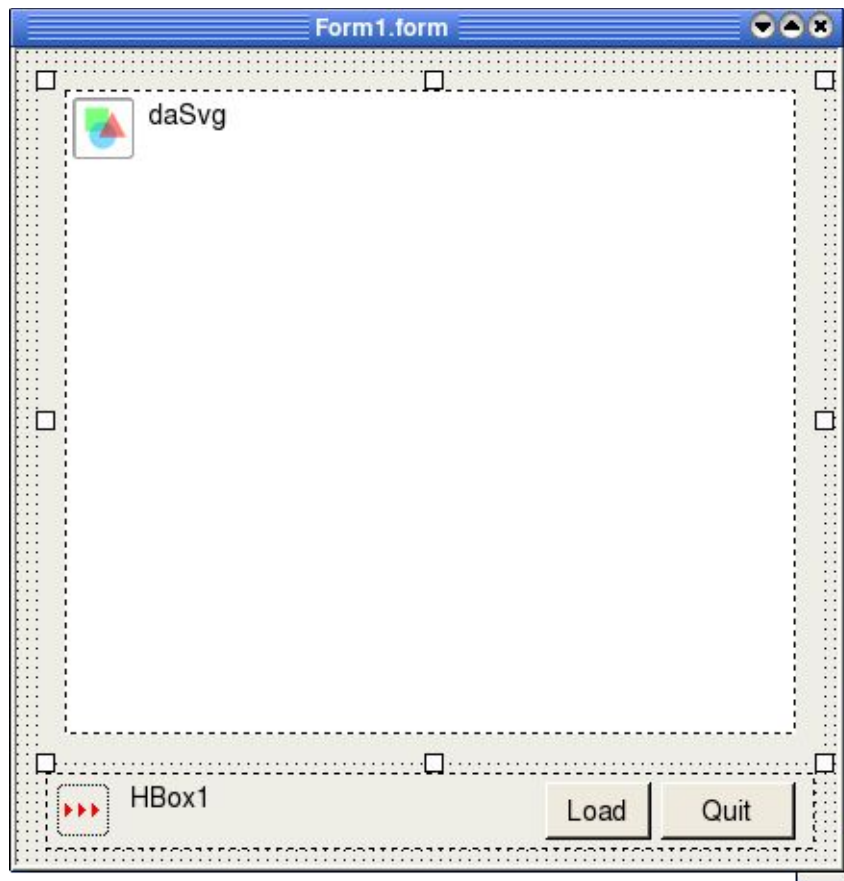
Double-click on the form when it opens and the code window appears. Start with the ScrollView control, naming it ScrollView1. Next, add a

DrawingArea on top of the ScrollView control. Name the DrawingArea daSVG. Add the HBox, named HBox1 and the two buttons named LoadBtn and QuitBtn. Don't forget to add the text captions to the buttons. Next, double-click on the Quit button and add this code:

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

Now, double-click on the LoadBtn and add this:

```
PUBLIC SUB LoadBtn_Click()

 'declare our Drawing object for the SVG file
 DIM dra AS Drawing

 'instantiate it
 dra = NEW Drawing

 'now load it using the Load method
 dra.Load("floppy.svg")

 'set cached property to TRUE to cache it in memory
 daSvg.Cached = TRUE

 'set visible property to TRUE so we can see the picture
 daSvg.Visible = TRUE

 'move the drawing area object to start at top left (0,0) origin
 dasvg.Move(0,0,dra.Width, dra.Height)

 'call the mandatory Draw.Begin to start to draw
 draw.Begin(daSvg)

 'do the draw
 draw.Drawing(dra,0,0)

 'call the Draw.End method to quit drawing
 draw.end

 'refresh our form to see what was done
 Form1.Refresh
END
```

Once you have downloaded or created the floppy.svg file, save it to the project directory. That is all the code we need to add to view an SVG file. Save

your work and run the program. When the program starts, click the Load button and you should see a picture similar to this:
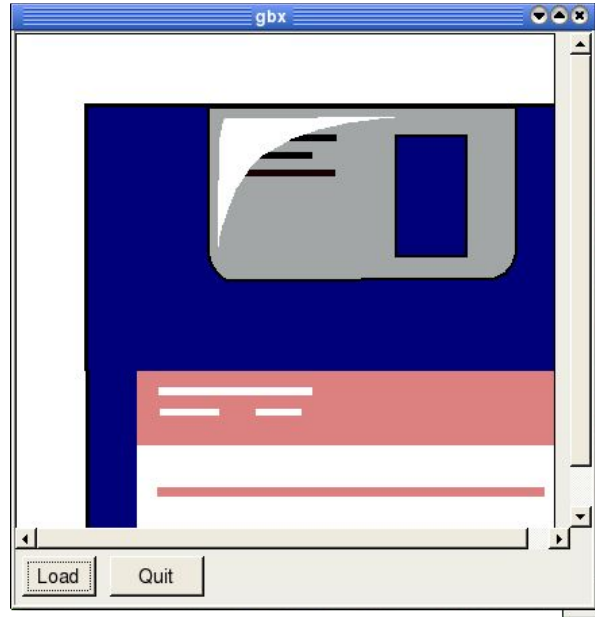
Figure 85- Loading an SVG file in Gambas.

I hope you have enjoyed our short little "*tour de force*" of Gambas drawing capabilities. This chapter has presented all of the basics needed to use Gambas for almost any graphics-related needs. At the end of this book, in Appendix A, we present a special graphics program that you can use to draw even more advanced 2D/3D rendered objects using a GKS-type graphics class created in Gambas.

A Beginner's Guide to Gambas

Contents of SVG file "floppy.svg" downloaded from http://openclipart.org:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<!-- Created with Sodipodi ("http://www.sodipodi.com/") --><svg height="400pt" id="svg548"
    sodipodi:docbase="/home/nicu/svg_gal/computers/" sodipodi:docname="/home/nicu/svg_gal/computers/floppy.svg"
    sodipodi:version="0.32" width="400pt" xmlns="http://www.w3.org/2000/svg"
    xmlns:sodipodi="http://sodipodi.sourceforge.net/DTD/sodipodi-0.dtd" xmlns:xlink="http://www.w3.org/1999/xlink">
 <metadata>
  <rdf:RDF xmlns:cc="http://web.resource.org/cc/" xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <cc:Work rdf:about="">
    <dc:title>Clipart by Nicu Buculei - antenna</dc:title>
    <dc:description></dc:description>
    <dc:subject>
     <rdf:Bag>
      <rdf:li>hash</rdf:li>
      <rdf:li></rdf:li>
      <rdf:li>computer</rdf:li>
     </rdf:Bag>
    </dc:subject>
    <dc:publisher>
     <cc:Agent rdf:about="http://www.openclipart.org">
      <dc:title>Nicu Buculei</dc:title>
     </cc:Agent>
    </dc:publisher>
    <dc:creator>
     <cc:Agent>
      <dc:title>Nicu Buculei</dc:title>
     </cc:Agent>
    </dc:creator>
    <dc:rights>
     <cc:Agent>
      <dc:title>Nicu Buculei</dc:title>
     </cc:Agent>
    </dc:rights>
    <dc:date></dc:date>
    <dc:format>image/svg+xml</dc:format>
    <dc:type rdf:resource="http://purl.org/dc/dcmitype/StillImage"/>
    <cc:license rdf:resource="http://web.resource.org/cc/PublicDomain"/>
    <dc:language>en</dc:language>
   </cc:Work>
   <cc:License rdf:about="http://web.resource.org/cc/PublicDomain">
    <cc:permits rdf:resource="http://web.resource.org/cc/Reproduction"/>
    <cc:permits rdf:resource="http://web.resource.org/cc/Distribution"/>
    <cc:permits rdf:resource="http://web.resource.org/cc/DerivativeWorks"/>
   </cc:License>
  </rdf:RDF>
 </metadata>
 <defs id="defs550"/>
 <sodipodi:namedview id="base" showgrid="true"/>
 <path d="M 52.5712 53.9929 L 407.003 53.9929 L 443.463 89.6056 L 441.768 447.429 L 53.4191 445.733 L 52.5712
    53.9929 z " id="path551" style="fill:#00007b;fill-rule:evenodd;stroke:black;stroke-opacity:1;stroke-width:3.75;stroke-
    linejoin:miter;stroke-linecap:butt;fill-opacity:1;stroke-dasharray:none;"/>
 <path d="M 371.39 53.9929 L 371.39 145.569 C 371.39 145.569 368.846 159.135 355.279 163.375 C 341.713 164.222
    156.866 164.223 156.866 164.223 C 156.018 165.071 144.995 157.439 144.147 152.352 C 143.299 147.264 142.451
    145.569 142.451 145.569 C 142.451 145.569 144.147 53.145 143.299 53.145 C 142.451 53.145 372.238 53.9929
    371.39 53.9929 z " id="path552" sodipodi:nodetypes="cccccccc" style="fill:#a3a6a6;fill-rule:evenodd;stroke:black;stroke-
    opacity:1;stroke-width:2.32865;stroke-linejoin:miter;stroke-linecap:butt;fill-opacity:1;stroke-dasharray:none;"
```

```
    transform="matrix(1.000000,0.000000,0.000000,1.152582,0.000000,-6.413116)"/>
<rect height="185.694977" id="rect553" style="font-size:12;fill:#ffffff;fill-rule:evenodd;stroke-width:1pt;fill-opacity:1;"
    width="321.362518" x="89.879768" y="249.862976"/>
<rect height="9.32714844" id="rect554" style="font-size:12;fill:#808080;fill-rule:evenodd;stroke-width:1pt;"
    width="0.00000000" x="410.394348" y="240.535797"/>
<path d="M 334.929 76.0389 C 334.081 76.0389 282.358 76.0389 282.358 76.0389 L 282.358 165.071 L 334.929
    165.071 L 334.929 76.0389 z " id="path555" style="fill:#00007b;fill-rule:evenodd;stroke:black;stroke-opacity:1;stroke-
    width:2.5;stroke-linejoin:miter;stroke-linecap:butt;fill-opacity:1;stroke-dasharray:none;"/>
<rect height="55.1149292" id="rect556" style="font-size:12;fill:#dd8080;fill-rule:evenodd;stroke-width:1pt;fill-opacity:1;"
    width="321.362549" x="89.8797607" y="249.862946"/>
<rect height="6.783386" id="rect557" style="font-size:12;fill:#dd8080;fill-rule:evenodd;stroke-width:1pt;fill-opacity:1;"
    width="287.445648" x="105.142365" y="335.503052"/>
<rect height="5.935455" id="rect558" style="font-size:12;fill:#dd8080;fill-rule:evenodd;stroke-width:1pt;fill-opacity:1;"
    width="287.445648" x="104.294426" y="371.115822"/>
<rect height="5.935455" id="rect559" style="font-size:12;fill:#df8080;fill-rule:evenodd;stroke-width:1pt;fill-opacity:1;"
    width="286.597717" x="105.142365" y="406.728516"/>
<rect height="5.08750916" id="rect567" style="font-size:12;fill:#000000;fill-rule:evenodd;stroke-width:1pt;fill-opacity:1;"
    width="78.8567657" x="159.409393" y="75.1909637"/>
<rect height="5.08754730" id="rect568" style="font-size:12;fill:#000000;fill-rule:evenodd;stroke-width:1pt;fill-opacity:1;"
    width="58.5066528" x="161.105225" y="87.9097748"/>
<rect height="5.08750916" id="rect569" style="font-size:12;fill:#180003;fill-rule:evenodd;stroke-width:1pt;fill-opacity:1;"
    width="76.3130035" x="161.105225" y="100.628624"/>
<rect height="5.93548584" id="rect570" style="font-size:12;fill:#ffffff;fill-rule:evenodd;stroke-width:1pt;fill-opacity:1;"
    width="114.469490" x="105.990288" y="261.733826"/>
<rect height="5.08752441" id="rect571" style="font-size:12;fill:#ffffff;fill-rule:evenodd;stroke-width:1pt;fill-opacity:1;"
    width="44.0919495" x="106.838211" y="277.844360"/>
<rect height="5.087512" id="rect572" style="font-size:12;fill:#ffffff;fill-rule:evenodd;stroke-width:1pt;fill-opacity:1;"
    width="33.916901" x="178.063675" y="277.844386"/>
<path d="M 150.317 158.228 C 151.899 153.481 148.734 63.2911 155.063 63.2911 C 161.392 63.2911 292.722
    61.7089 292.722 61.7089 C 232.595 68.038 164.557 66.4557 150.317 158.228 z " id="path566"
    sodipodi:nodetypes="cccc" style="fill:#ffffff;fill-rule:evenodd;stroke:none;stroke-opacity:1;stroke-width:1pt;stroke-
    linejoin:miter;stroke-linecap:butt;fill-opacity:0.5;"/>
</svg>
```

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, it's 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 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 un-noticed 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 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:
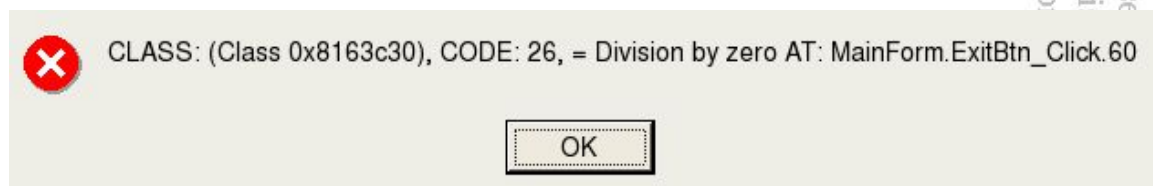
Figure 86- Error results caught with TRY.

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 mens 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 Form1. Add two buttons, Error and Quit, named ErrBtn and QuitBtn. Place the buttons on the bottom right corner of the form. Create a TextLabel 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 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:

Figure 87- CATCH test program opening screen.

Now, click the error button to generate our error and you should see this:
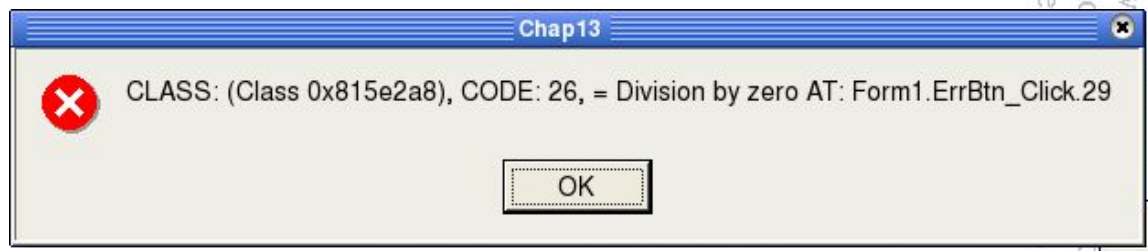
Figure 88- Div by Zero error caught by CATCH.

You should also notice that the main screen TextLabel has changed to this:

Figure 89- The TextLabel updated with error info.

Once you click the Ok button for the Error Message dialog, you will see this:
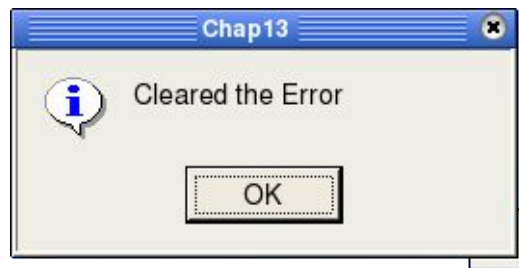
Figure 90- Info message that the error is cleared.

Now, click the OK button on the Info message and the TextArea is updated like this:

Figure 91- 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:

```
' 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:
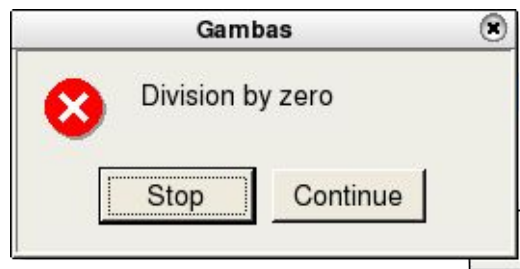
Figure 92- Default Gambas error dialog.

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.

A unique feature of Gambas is that an object can raise an event. In order to raise an event, the object must be given a name (or handle) at the time it is created with the NEW keyword. Without being assigned a name, events raised by an object are ignored. Events are caught by the parent of an object. The parent is actually an object's unique event observer. The parent acts as the default observer for the object when an event is raised. This parent can be a class, in which case the event handler must be a static method. In this case, the handle and the parent both can be changed at runtime using the Object.Attach() and Object.Detach() methods of the Object class. The handle of the object is not stored with an object and therefore cannot be retrieved.

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 Event1Btn_Click()
DIM bResult AS Boolean

  RAISE MyEvent("Raise an event") AS bResult

  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:
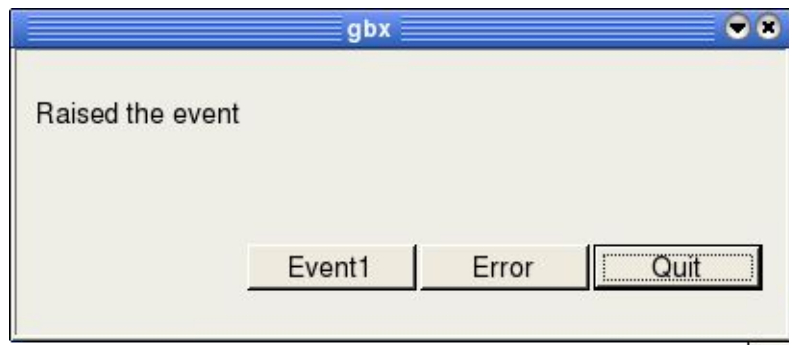
Figure 93- Our event was raised.

Chapter 14 – Mouse, Keyboard and Bit Operations

The mouse was designed to be a replacement for gathering keyboard input from a user. 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      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 while **X** and **Y** return the 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, Form1 and make it a startup class form. Next, add a drawing area named _da_ to the top left corner of the Form1 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
   '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:
```
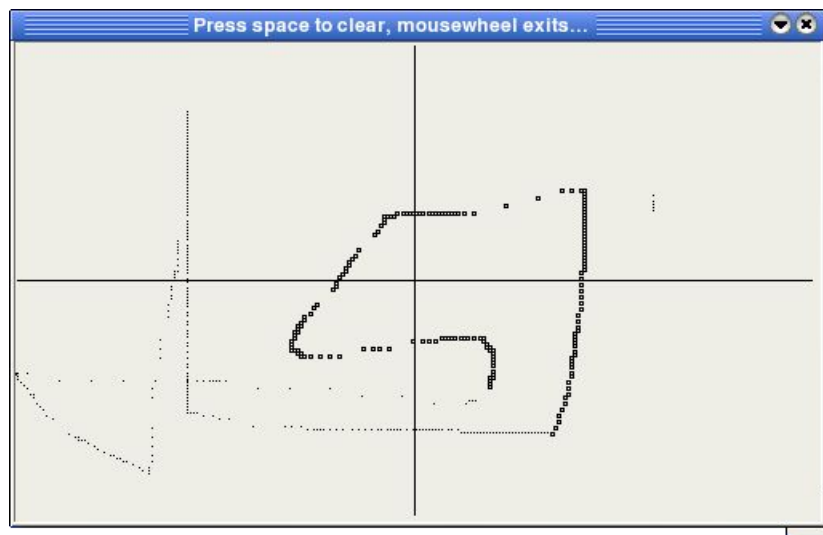

Figure 94- MouseOps program running.

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	**Esc**	**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, Form1 and make it 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:

```
' 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
```

As you can see, the code above is pretty straight-forward and simple. When you run the program, you should see something like this:
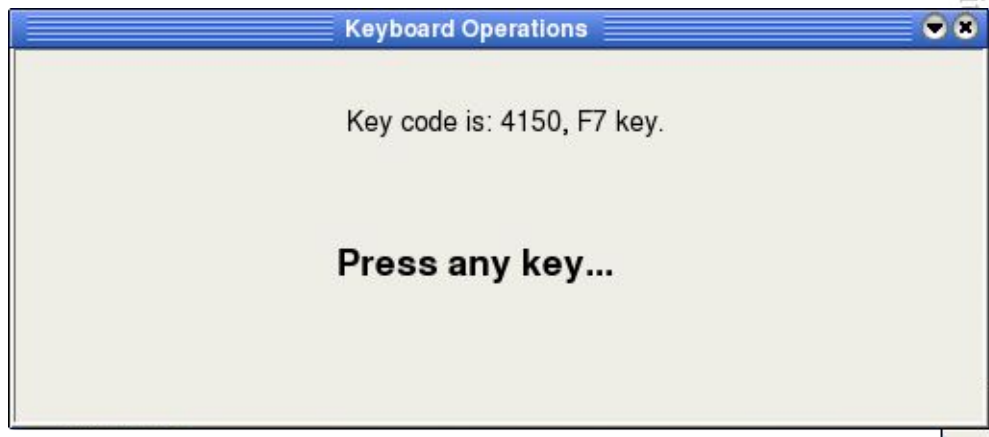
Figure 95- KbdOps program running.

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 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 11111111111111111111111111111111 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 *operand1* using *operand2*
BClr	Value = BClr(Number , Bit)	Clear a specific bit in *operand1* using *operand2*
BSet	Value = BSet(Number , Bit)	Return TRUE if the the bit specified in *operand2* of *operand1* is set.
BChg	Value = BChg(Number , Bit)	Returns *operand1* with the bit specified in *operand2* inverted.
Shl	Value = Shl(Number , Bit)	Returns *operand1* with the bit specified in *operand2* shifted left one bit position.
Shr	Value = Shr(Number , Bit)	Returns *operand1* with the bit specified in *operand2* shifted right one bit position.
Rol	Value = Rol(Number , Bit)	Returns *operand1* rotated left *operand2* bit positions.
Ror	Value = Ror(Number , Bit)	Returns *operand1* rotated right *operand2* 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, Form1 and make it a 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, named TextLabel1 through Textlabel7. We will need two TextBoxes, TextBox1 and TextBox2. Finally, we will add nine buttons named RolBtn, RorBtn, ShlBtn, ShrBtn, ClearBtn, TestBtn, SetBtn, ChangeBtn, and the QuitBtn. Layout the controls

above on the form, as shown in the figure below:
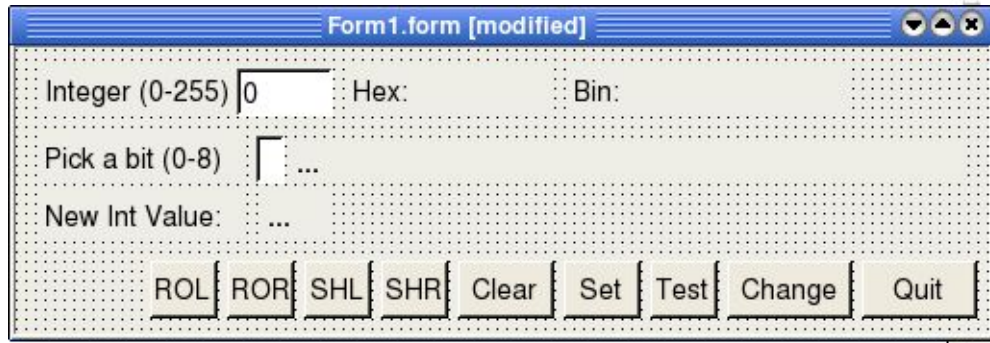
Figure 96- The BitOps program in design mode.

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 = " bitOpns "
END
```

Now, the program will display the caption " bitOpns " 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()
 form1.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
     Label2.Text = "Hex: " & Hex(myInt) 'convert hex value
     label3.Text = "Bin: " & Bin(myInt) 'convert bin value
     label7.Text = Str$(myInt) 'update our text fields
     form1.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 = BClr(myInt, result)
     'update the label displays
     label5.Text = "Bclr: " & Bin(i)
     label7.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
      label5.Text = "Bset: " & Bin(i)
      label7.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)
     label5.Text = "Bchg: " & Bin(i)
     label7.Text = Str$(i)
   ENDIF
END
```

The Shl function is 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)
     label5.Text = "SHL: " & Bin(i)
     label7.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)
     label5.Text = "SHR: " & Bin(i)
     label7.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
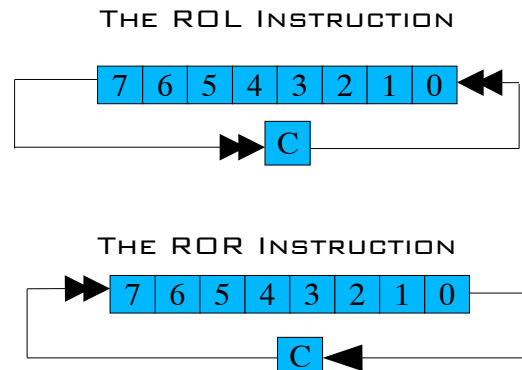
ROL and ROR work such that the carry bit is shifted into the end bit that is left empty by the rotation process instead of inserting a zero value. ROL shifts the contents of a byte one place to the left. ROL does not place a zero into bit 0. Instead, it moves the carry bit from bit position 7 into bit 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)
     label5.Text = "ROL: " & Bin(i)
     label7.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)
     label5.Text = "ROR: " & Bin(i)
     label7.Text = Str$(i)
  ENDIF
END
```

If you want your program to check 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)
     label5.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:
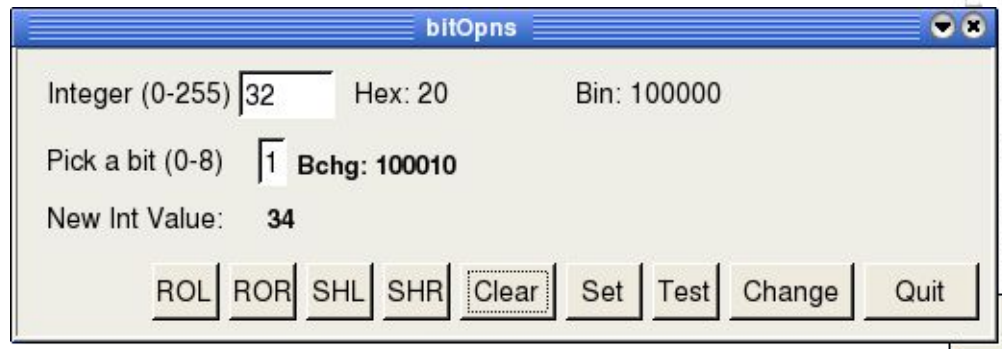
Figure 97- Our bitOps program running.

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 basic 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 – Gambas and Databases

This chapter describes in detail the process of using a MySQL or PostgreSQL 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 site[18]. At a minimum, complete the tutorial in Chapter 3 of the MySQL User Manual before proceeding with this chapter. Once you are

---

18 See URL http://www.mysql.com/ .

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 management systems: PostgreSQL, MySQL and SQLite. 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 *Connection, DB, Database, Field, Index, Result, ResultField, Table,* and *User*.

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, Databases, Host, Login, Password, Name, Port, Tables, Type, Users and Version. We will explain each of these in the following sections.

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 can 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.

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 sytstem. 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 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
```

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

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 databasetype are: "postgresql", "mysql", and "sqlite". The type property uses this syntax:

```
PROPERTY Type AS String
```

Users

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. Within MySQL, transactions are only known with "*transaction safe*" tables (i.e., BDB and InnoDB). The other table types COMMIT immediately. 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 **A**tomicity, **C**onsistency, **I**solation and **D**urability (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 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 tblDEFAULT where id = "
' we will insert a parameter value (12) at the end of the query string
iParm = 12

$hConn.Begin
MyResult=$hConn.Edit("tblDEFAULT", sQuery, iParm)
MyResult!Name = "Mr Rittinghouse" ' Set a field value
$hConn.Update ' Update with the new value
$hConn.Commit ' make the changes permanent
$hConn.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.

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 query
' sDBTable 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
```

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 andType.   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, and Length. 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.

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, Edit, Exec, Find, Open, Quote and Rollback) 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.

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. Lentgh 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.

User

The User 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 User class has a single method, Delete which is used to delete a user from the 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 that we are going to explore. Written by Gambas founder Benoit Minisini, this program is an excellent example of how to code your program to use database within the Gambas environment.   To begin, start Gambas and choose the Database program from the Examples menu. When the IDE appears, you will see that the program contains two forms, FMain and FRequest. FMain should look like the figure below.

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.
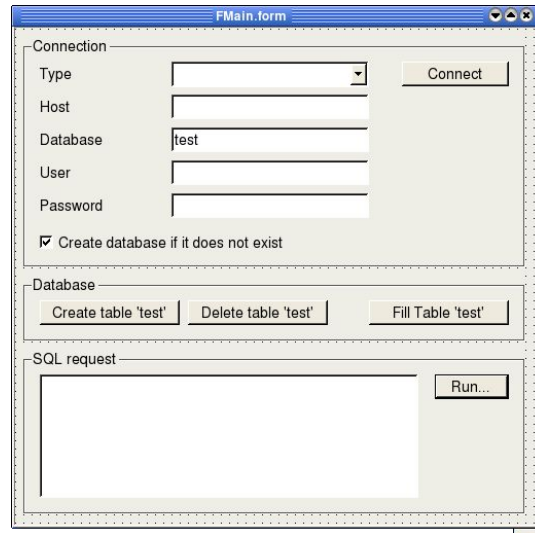
Figure 98- The FMain form in design mode.

The FRequest form is much simpler to build. It is shown in the figure below:
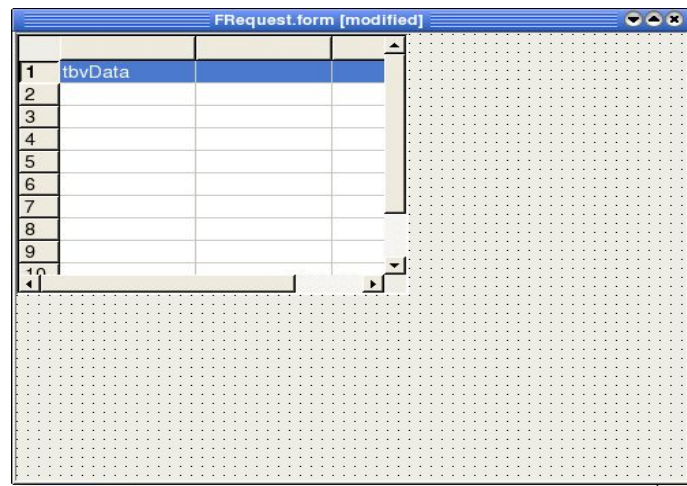
Figure 99- FRequest Form in design mode.

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.

CATCH
  Message.Error(Error.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. I have modified it slightly to avoid using hard-coded strings and to use the same name as the database. The original code is commented out so you can use either method.

```
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("test")

  'this is the code I changed to avoid hard-coding the string
  'as was done in the previous line of code
  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

'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 following segment of code was added by the author to enable or disable 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(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. I have also

modified it slightly to avoid using hard-coded strings and to use the same name as the database. The original code is commented out so you can use either method.

```
PUBLIC SUB btnDelete_Click()

  'remove the table
  '$hConn.Tables.Remove("test")
  $hConn.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
  '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 server 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)

  '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
  $hConn.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, in Gambas, it is still usable and we will show you how it is done. Be aware, however, that it is *not recommended for use* as Trolltech, the owners of the QT library, have publicly declared the TableView control obsolete. The Gambas TableView is based upon QTable and QtableItem, both of which will become obsolete in Qt 4.0. TableView will more than likely then be based upon QTableView or QTableWidget instead.

```
PRIVATE SUB ReadData()
  'this variable is declared but never used, you can comment it out
  DIM hTable AS Table

  DIM hField AS ResultField

  'this variable is declared but never used, you can comment it out
  DIM sField AS String
  DIM iInd AS Integer

  'this variable is declared but never used, you can comment it out
  DIM iLen 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

   '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 Qt 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
  $rData.MoveTo(Row)
  'this actually assigns data from the results object to the tableview
  tbvData.Data.Text = Str($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
```

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 of globalization that involves two processes, internationalization (I18N) and localization (L10N).

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 ramifications and 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 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. In this manner, an internationalized application can be *localized* very 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 process is translation of text. Other types of data, such as sounds and images, may require localization if they are culturally sensitive. Localization engineers verify 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 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 Characters in Unicode are identified by their number or "code point" which is usually given in hexadecimal notation. For example, in Hebrew, the letter "he" is code point 5D4 which is usually written with the convention of prefixing a "U+" in front of the code point and padding it with leading zeros as needed, i.e., U+05D4.

Unicode

Unicode currently defines just under 100,000 characters, but has space for 1,114,112 code points[19]. The code points are organized into 17 "planes" of 2^{16} (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, etc.) and across India onward 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.

---

19 The reader is encouraged to visit http://www.tbray.org/ongoing/When/200x/2003/04/26/UTF for a full explanation.

ASCII files can be converted into a UCS-2 file by inserting 0x00 in front of every ASCII byte. To make a UCS-4 formatted file, you must insert three 0x00 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 Thompson[20]. 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 history[21]). 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 10646-1[22]. Within a week of that time, 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 Conference[23]. 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).

---

20 http://www.cs.bell-labs.com/who/ken/

21 http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt

22 http://64.233.167.104/search?q=cache:U4zV7dcaq0EJ:www.faqs.org/ftp/rfc/pdf/rfc2279.txt.pdf+Greger+Leijonhufvud&hl=en

23 USENIX Technical Conference held in San Diego, CA, January 25-29, 1993, published in the Conference Proceedings, pp. 43-50.

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., it's code position, code point or Unicode scalar value*). This encoding form has the following characteristics[24] (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.

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.

---

24 See URL http://www.faqs.org/rfcs/rfc2279.html.

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 0x00 to 0x7F (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 (0x00-0x7F) 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 0xC0 to 0xFD 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 0x80 to 0xBF. 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 0xFE and 0xFF 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: 0xxxxxxx
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:
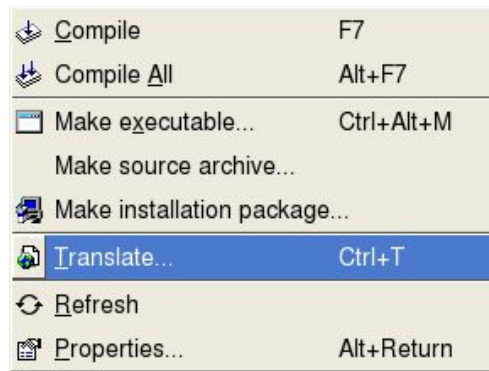
Figure 100- Choosing the Translate option in Gambas.

Next, you must select the target translation language in the combo-box of the translate dialog box, as shown below:
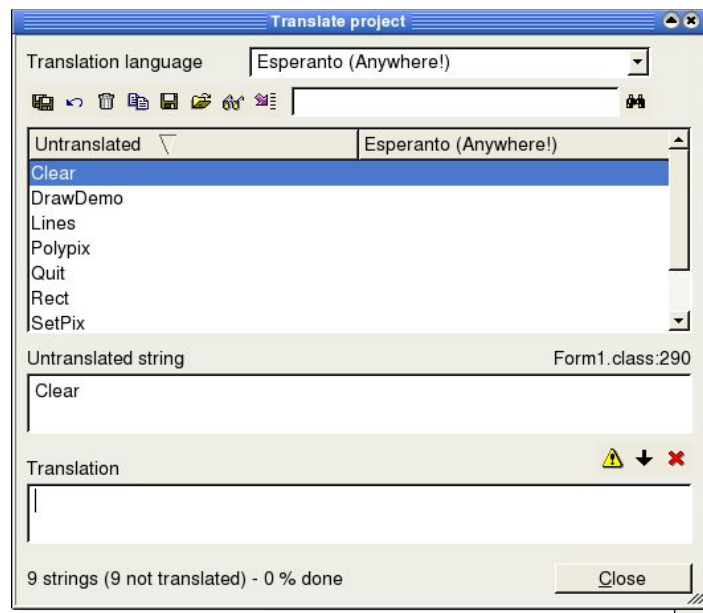
Figure 101- The Translate Dialog in Gambas.

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: 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 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 untranslated strings in your program could change. However, any strings you have translated strings will be preserved. If an untranslated string disappears completely, the translation will also disappear and if a new untranslated string is added, it gets an empty or null translation string assigned to it. All other untranslated 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.,[25] you should consider creating an I18N testing checklist for your application. Some of the questions they recommend you consider include the following:

1. Does program design account for international considerations?

2. Can a user enter text, accelerators, and shortcut-key combinations on an international keyboard?

3. Can a user enter dates, times, currencies, and numbers using international formats?

4. Can a user successfully cut/paste text with accented (or double-byte) characters?

5. Can the application work correctly on different types of hardware sold in the target market?

6. Does the application work correctly on localized operations systems?

7. Can a user type European accented (or Asian double-byte) characters in documents and dialog boxes?

8. Can documents created in a target language be opened in other languages and vice-versa?

9. Can users save and 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

---

25 Huang, E., Haft, R. and Hsu, J. "Developing a Roadmap for Software Internationalization", Whitepaper dated October 2000, retrieved fromURL www.symbio-group.com/doc/Developing a Roadmap for Software Internationalization.pdf on October 5th, 2005.

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.

This page intentionally blank.

Alphabetical Index

A Beginner's Guide to Gambas