

# Getting Started with Gambas Version 2: A Tutorial

Subject: Learning Visual Basic with Linux and Gambas  
Author: Timothy Marshal-Nichols  
File Name: Getting-Started-with-Gambas-Tutorial.odt  
Version: 1.0 (Revision: 193) Sunday 28 May 2006



## Contents

1: Introduction.....	2
1.1: GAMBas Almost Means Basic.....	2
1.2: Projects.....	4
1.3: Gambas Resources.....	5
1.4: License.....	6
1.5: Acknowledgments.....	6
2: Your First Gambas Project: SimpleEdit.....	7
2.1: Creating the project.....	7
2.2: Creating the user interface.....	13
2.3: Adding the code.....	18
2.4: Running the project.....	23
3: Drawing Project: ImageShow.....	28
3.1: Creating the project.....	28
3.2: Creating the user interface.....	29
3.3: Adding the Code.....	37
3.4: Running the project.....	52
4: Database Project: Notations.....	56
4.1: Creating the project.....	59
4.2: Creating the user interface.....	64
4.3: Checking our user interface for CRUD.....	75
4.4: Adding the code.....	77
4.5: Running the project.....	99
4.6: Switching to a MySQL or PostgreSQL Database.....	102
5: Appendix 1: Database Commands with Exec and SQL.....	104

## 1: Introduction

This is a simple Getting Started with Gambas Tutorial. It is intended for first time programmers who want to gain some idea of the capabilities of Gambas. You can then see if this is the development environment that they would like to learn more fully. It will also be useful for Visual Basic programmers who are moving from Windows to Linux.

Traditionally on Unix based systems like Linux you used a number of tools to develop an application. So you used a code editor, possibly a user interface designer, a compiler and other tools. You performed each stage of the development process with the relevant tool. This process can work, after all much of Linux was built this way. It does have the advantage that if you do not like one tool then you can replace just that tool with something you do like. But it can also have the disadvantage of compatibility issues when the output of one tool does not fit the required input for another tool. It also makes it difficult for new developers to learn the required steps to produce professional looking application. To overcome these problems there are now a number of Integrated Development Environment (IDE) for Linux. IDE's include all the tools you need to develop applications within one framework. They have been very popular on other operating systems and developers moving to Linux expect to find similar IDE's. Two of the most used IDE's on Linux are KDevelop and MonoDevelop. KDevelop is an IDE for creating C and C++ applications and for using scripting languages. MonoDevelop is an IDE for developing .NET application using languages like C# and VisualBasic.NET and others.

Gambas is a IDE for Visual Basic on Linux. You can build most kinds of Linux application with Gambas. However Gambas is especially strong at providing Graphical User Interfaces (GUI) types of applications. They can be stand alone applications or front ends to a server or database. Gambas provides all the tools you would normally expect in a IDE. It has a form designer where you can drag controls and components onto your forms to develop your user interface. It also has a project manager, a code editor, a code explorer and an integrated help system. You can also compile, run and debug your applications from within Gambas. It is open source and so fits in with the Linux philosophy.

Gambas also has a strong user community. There are a number of useful forums where new users can get help. There is an active Gambas mailing list. If there are bugs with Gambas then, in my experience, they are rapidly corrected. See the [Gambas Resources](#) section below.

Gambas is definitely a Visual Basic for Linux. It is not simply a port of Microsoft Visual Basic to Linux. However Windows users of Visual Basic (at least up to version 6) will find much in this environment that is familiar.

### 1.1: *GAMBas Almost Means Basic*

BASIC stands for Beginner's All Purpose Symbolic Instruction Code. This language was developed by J. Kemeny and T. Kurtz at Dartmouth College to teach beginner programmers. Ever since it has

proved to a useful first language for beginners to learn. BASIC became very popular when personal computers (PC's) first appeared in the 1970's. The small overheads of the language and the ease of learning made it the language of choice for new programmers with their first PC. Microsoft understood this and shipped QuickBASIC for MS-DOS.

Alan Cooper is credited as the 'father' of Visual Basic and sold the idea to Microsoft. Microsoft also learned from HyperCard on the Mac and in 1991 launched Visual Basic 1.0. But it was not until Windows 3.0 and Visual Basic 3.0 that this really became popular. Since then it has become one of the most dominant development environment, both in the home and in industry.

There have been versions of the BASIC language for almost all operating systems and Linux is no exception. There have been QuickBASIC like Unix/Linux versions. There are also versions of Visual Basic for Linux like HBasic and KBasic. For more information and examples see <http://www.thefreecountry.com/compilers/basic.shtml>. Arguably the most developed and usable of the Visual Basics for Linux is Gambas.

*Gamba* is the Spanish for “prawn”. Gambas is also an Visual Basic IDE designed by Benoit Minisini. The first public release of Gambas was Version 0.20 in February 2002. There then followed a great deal of development work and releases. The first stable release of Gambas was Version 1.0 in January 2005. Since then there have been a number of bug fixes to the stable version. Also in January 2005 came the release of the new development version of Gambas (version 1.9.1). This version will eventually become the stable version 2.0.

All version of Gambas are issued under the GNU General Public License and are free to use. It also means the Gambas runtime is free to use if you need to install it on a customers workstation. As Gambas is open source you can obtain, look at and, if you need to, modify the source code. Also the Gambas IDE is written in Gambas.



In this tutorial we are using the latest development version of Gambas. At the time of writing the latest stable version is 1.0.16 (2 May 2006) and the latest development version is 1.9.28 (29 April 2006).

Most of what is written here should work on the stable version of Gambas. Where changes are needed you shall see a note like this. Look for the symbol on the left.

Also in the example projects that are part of this tutorial there are versions for Gambas 1 and 2.

All of the screen shots in this tutorial are from Gambas version 2. Some of the screens may look slightly different in Gambas version 1 or newer installs of version 2.

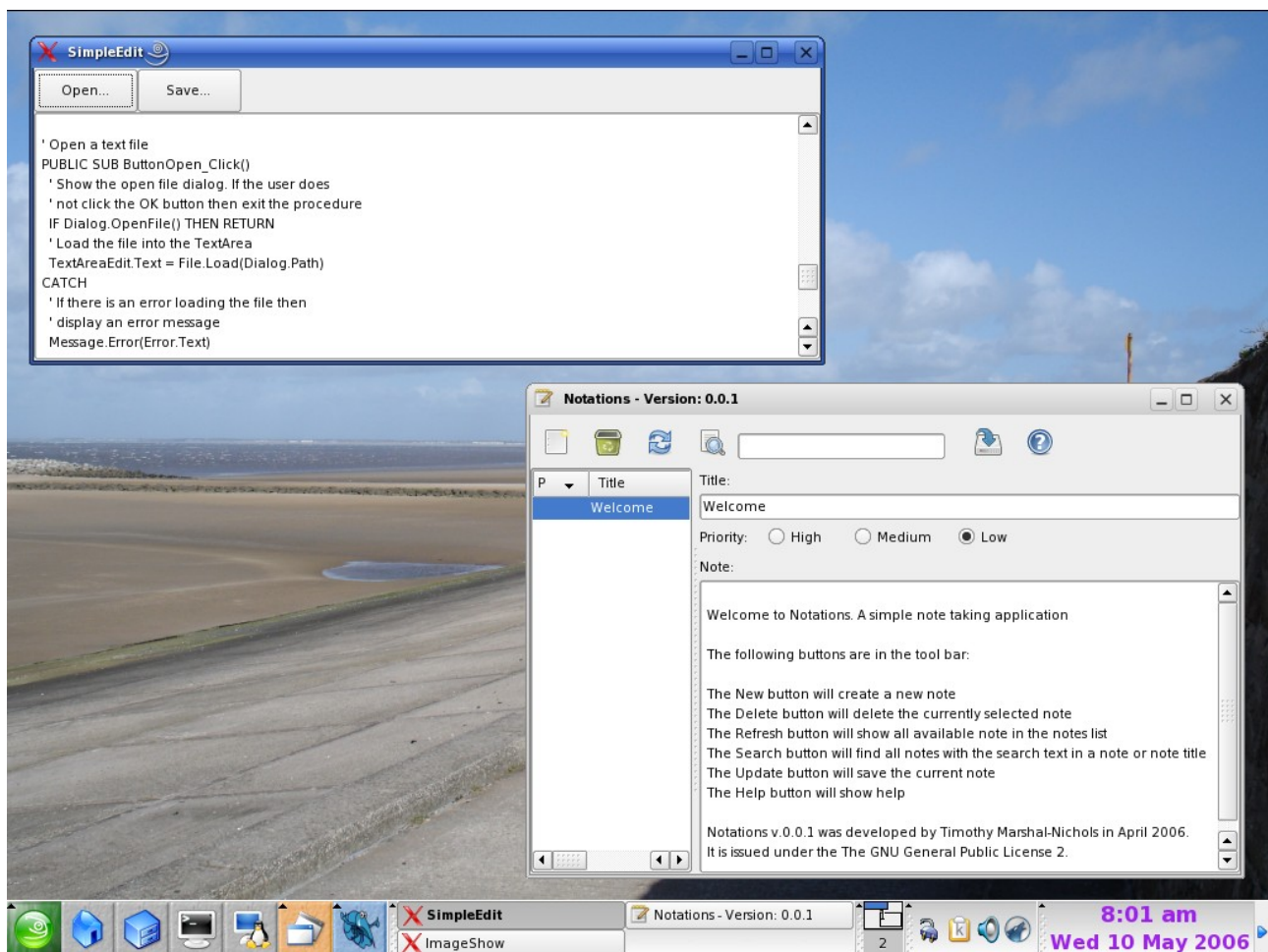
I have been using the development version of Gambas for about six months at the time of writing this tutorial. I would say that Gambas is more stable than Microsoft Visual Basic 6. I have had fewer crashes with the development version of Gambas than with production release of VB6. It is properly just as stable as Microsoft Visual Basic .NET (both the .NET runtimes 1.0 and 1.1) which I have also used quite extensively.

## 1.2: Projects

In this tutorial we shall develop three projects that demonstrate some of the types of applications that can be built with Gambas.

- [SimpleEdit](#). A simple text editor. This is a simple first application to demonstrate using the development environment. It will show how by using some of the components from the tool box and with a little coding you can build a usable application.
- [ImageShow](#). A image display program for playing a slide show of images from a selected directory. This application demonstrates some of the drawing functions in Gambas.
- [Notations](#). A project for editing and storing user notes. This project demonstrates how to create a user interface to a database. The application will also create the database if it does not exist. You can then create, read, update and delete records in the database.

The following screen shot shows all three applications in action.



One issue I am not going to deal with in this tutorial is installing Gambas itself. Installing applications on Linux can become hefty chapter in itself. Look at the [Resources](#) section of this introduction for links where you can find more help and information.

### 1.3: Gambas Resources

These are some of the web sites where you can find more information about Gambas. You may need some of these links to complete this tutorial if your Linux distribution has not included all the required resources. New sources of information about Gambas are coming on-line all the time. Indeed I have had to add items to this list a number of times while writing this tutorial. I just keep discovered more information about Gambas on the web. As always with web links they can become out of date fast – these were valid in May 2006.



#### Gambas:

The Gambas shrine	<a href="http://gambas.sourceforge.net/">http://gambas.sourceforge.net/</a>
Sourceforge page for Gambas download	<a href="http://gambas.sourceforge.net/download.html">http://gambas.sourceforge.net/download.html</a>
The Gambas help Wiki	<a href="http://www.gambasdoc.org/">http://www.gambasdoc.org/</a>
Gambas Wiki Book textbook	<a href="http://en.wikibooks.org/wiki/Gambas">http://en.wikibooks.org/wiki/Gambas</a>
German Gambas Wiki Book	<a href="http://de.wikibooks.org/wiki/Gambas">http://de.wikibooks.org/wiki/Gambas</a>
You can subscribe to the Gambas mailing list here:	<a href="https://lists.sourceforge.net/lists/listinfo/gambas-user">https://lists.sourceforge.net/lists/listinfo/gambas-user</a>
Italian Gambas site	<a href="http://www.gambas.it/">http://www.gambas.it/</a>



#### Gambas Forums:

Linux Basic	<a href="http://www.linuxbasic.net/">http://www.linuxbasic.net/</a>
My Gambas Community	<a href="http://forum.stormweb.no/">http://forum.stormweb.no/</a>
German Gambas Club	<a href="http://gambas-club.de/">http://gambas-club.de/</a>
Sitio web de Gambas (In Spanish)	<a href="http://gambas.gnulinex.org/">http://gambas.gnulinex.org/</a>



#### More Links:

SQLite	<a href="http://www.sqlite.org/">http://www.sqlite.org/</a>
MySQL	<a href="http://www.mysql.org/">http://www.mysql.org/</a>
PostgreSQL	<a href="http://www.postgresql.org/">http://www.postgresql.org/</a>
Tango Desktop Project	<a href="http://tango-project.org/">http://tango-project.org/</a>
GNU Licenses	<a href="http://www.gnu.org/licenses/">http://www.gnu.org/licenses/</a>
Mono Visual Basic.NET	<a href="http://www.mono-project.com/Language_BASIC">http://www.mono-project.com/Language_BASIC</a>
Free BASIC Compilers and Interpreters	<a href="http://www.thefreecountry.com/compilers/basic.shtml">http://www.thefreecountry.com/compilers/basic.shtml</a>

## **1.4: License**

You are free to use the content of this tutorial and example source code as you wish. Also you can distribute this tutorial to whoever you wish. I would request that you acknowledge the author and also make sure the source code for example applications is freely available (both for Versions 1 and 2 of Gambas). But you are not forced to this. The text of this tutorial is issued under the **The GNU Free Document License**. All of the applications and code examples from this tutorial are issued under the **The GNU General Public License 2**. For details see <http://www.gnu.org/licenses/>.

## **1.5: Acknowledgments**

Many thanks to Rohnny Stormo at [My Gambas Community](#) for checking over this tutorial.

---

## 2: Your First Gambas Project: SimpleEdit

Our first project is going to be a simple text editor. As a starting point this may seem a little ambitious for someone who has never programmed in Gambas before. Admittedly our text editor is not going to be as full featured as some editors you may have used. But still it will be a usable text editor. This project shows the power of a modern development environment. By using the functionality of pre built components it is easy to build some usable applications quickly.

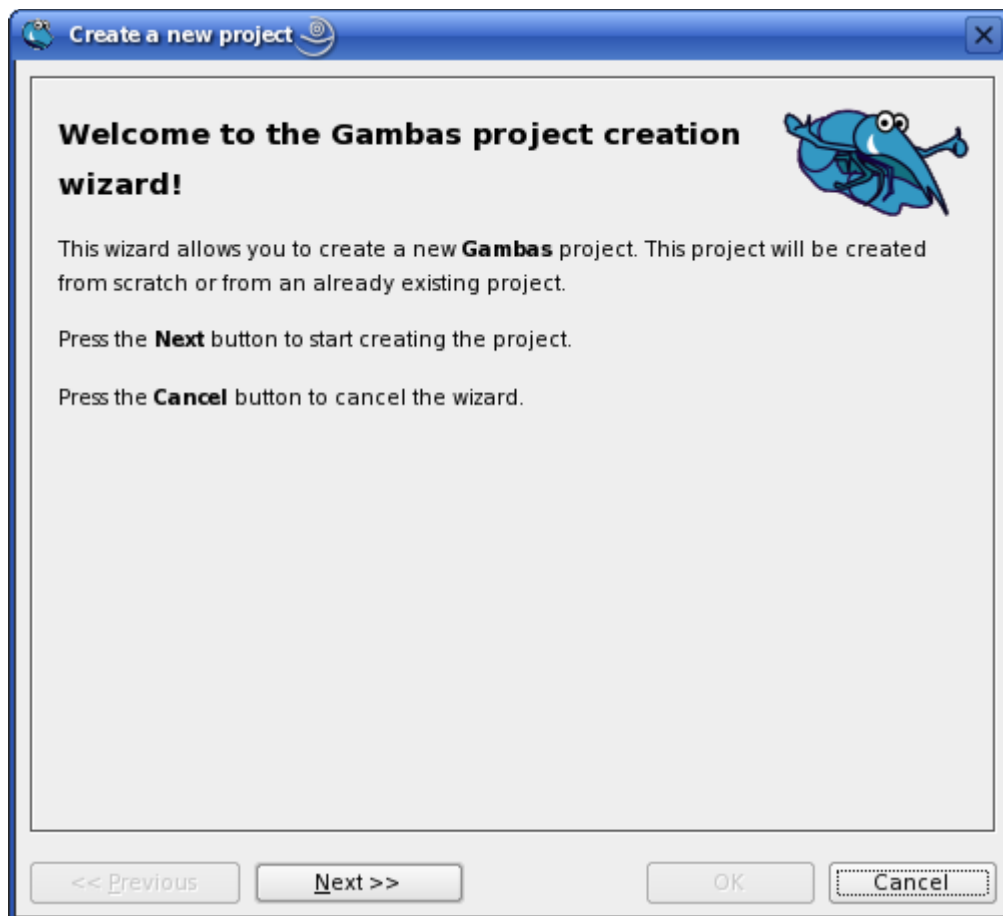
With SimpleEdit you can do many of the things you can do with any other text editor. You can open and save text documents. You will have to write a bit of code to achieve this. But the TextArea component we are going to use also gives us a lot of functionality. This includes being able to cut, copy and paste from the clipboard. It also gives us an undo facility. And we do not have to write any code to include this functionality in our application.

In this first project for Gambas we are going to provide detailed screen shots for most of the steps. In the later example projects we shall assume you do not need so many screen shots. You can always refer back to this project.

### ***2.1: Creating the project***

The first step is to create a new Gambas project. So open Gambas and select **New project...** This will start the new project wizard. The first page of the wizard simply shows a welcome screen with details of how to use the new project wizard.

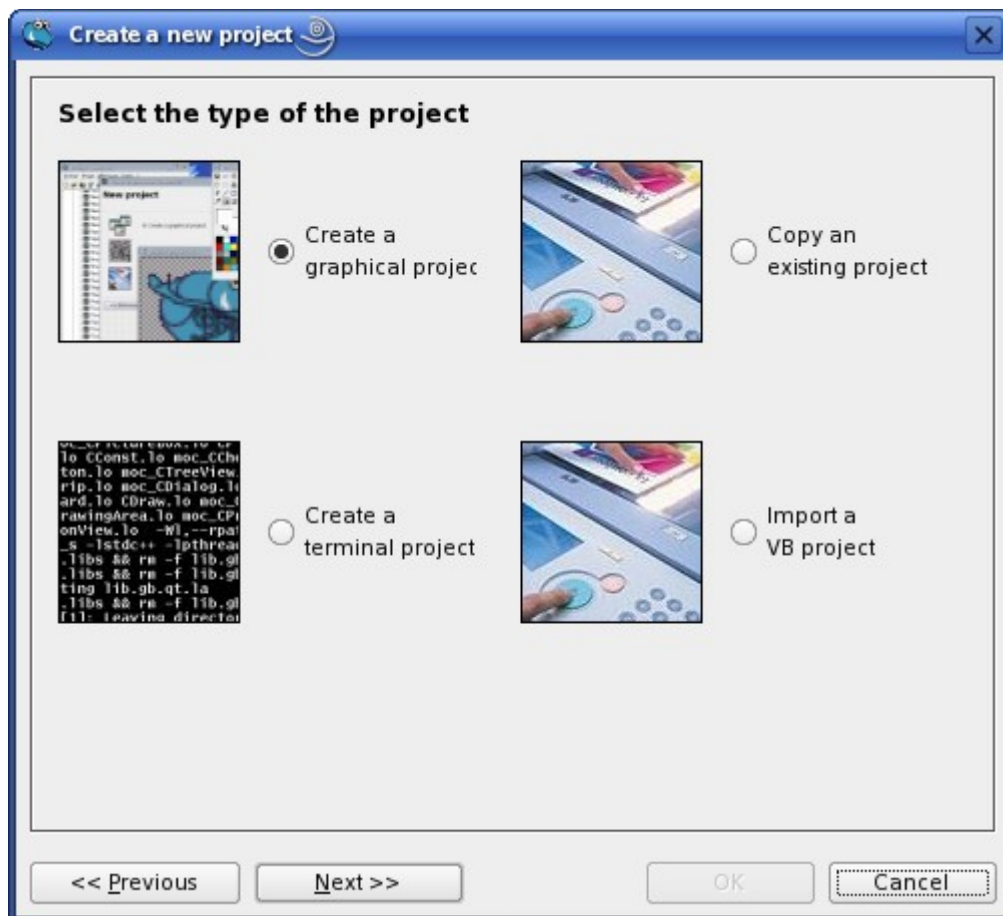




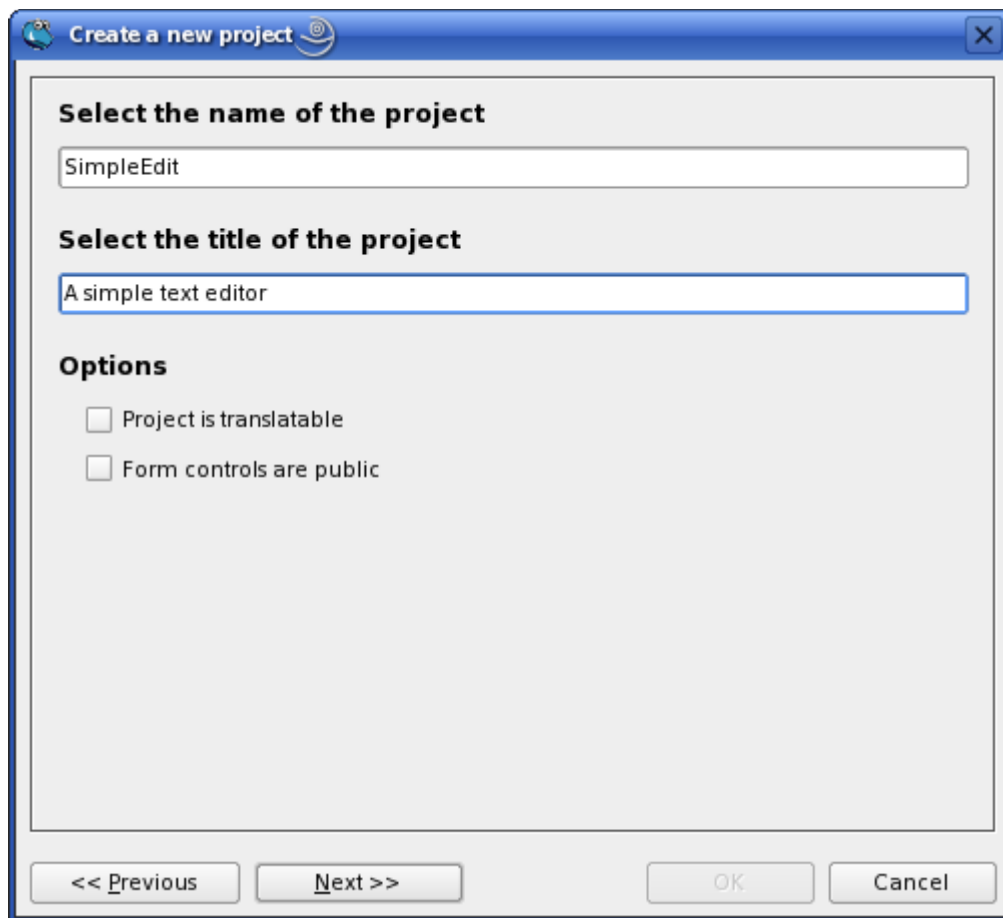
Click the **Next >>** button for the next page of the wizard. This shows a page where you can select the type of Gambas project you want to build. The types of projects include:

- **A graphical project.** As this is the most often used option and it is the default option. This type of application would have windows with which the user interacts. It would be controlled by the mouse and/or the keyboard.
- **A terminal project.** You would run this type of project from a terminal window. You can read and write data to standard input, error and output. It will accept parameters from the terminal and all the other things you would expect from a command line application.
- **Copy an existing project.** You used this option if you want to clone an existing project to use as the starting point for a new Gambas project.
- **Import a VB project.** Here you can import a project created with Microsoft Visual Basic.





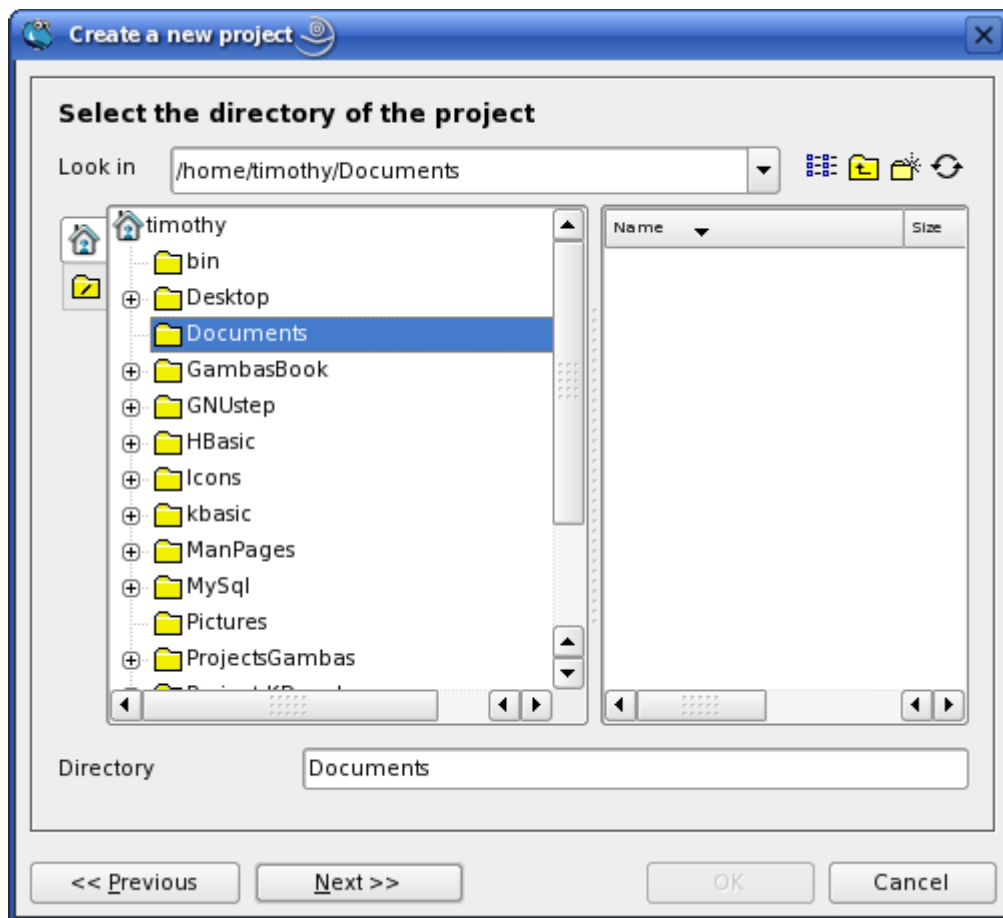
We are going to use the default option of a graphical project. So click the **Next >>** button. This shows a page when you enter the name and title for your project we are going to create.



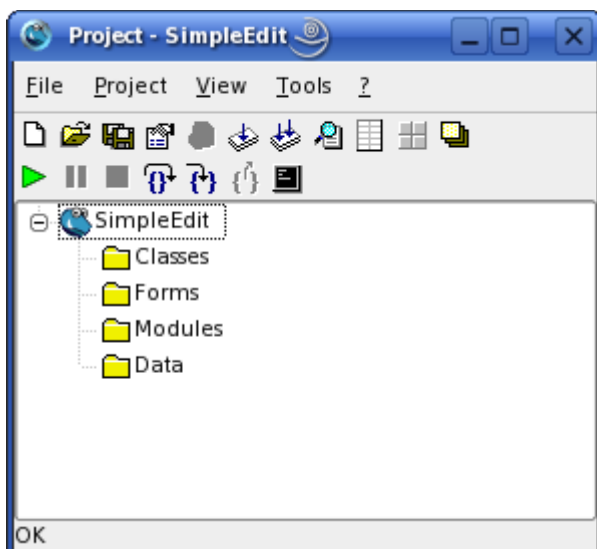
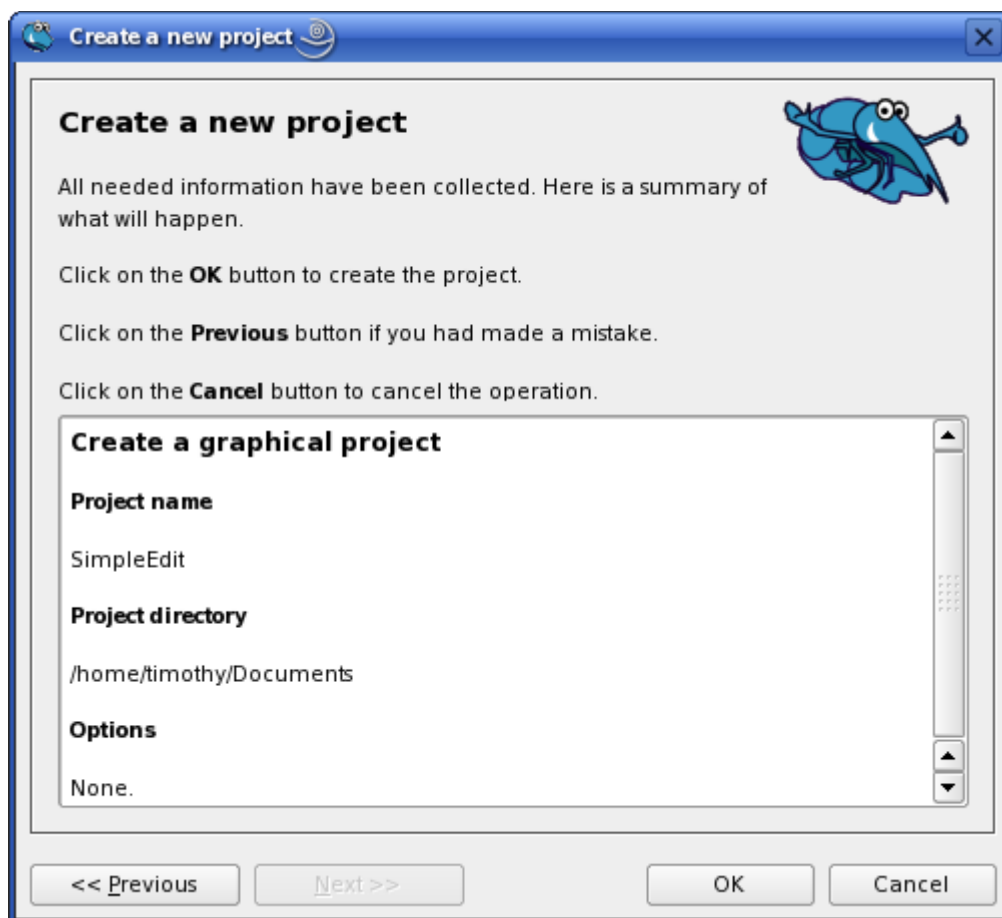
Enter the name of SimpleEdit and the title of A simple text editor. Leave the other options unchecked. Click on the **Next >>** button. This shows a page where you select the directory where you want to save the project files.



The following screen shot shows saving your project to the documents directory. This is simply to illustrate the dialogue. However it might be easier to set up a directory where you save all your Gambas projects. You could then store all your project in one location. This would then make managing your projects easier.



Select the location where you want to save the project. Then click the **Next >>** button. This final page of the wizard lists the options you have selected for the project.



Check through the options and then click the **OK** button. We have created a new Visual Basic project and Gambas will open showing the Gambas project manager.

Gambas will also open two other windows. One is a Toolbox. This shows some of the controls we can use in our projects. The other is the Properties window. This is where the properties of controls in your project can be edited. As we have not created any controls yet it will be blank.



If this is the first time you have run Gambas you may see two other items. The first is a Tip of the day window and the other is the Gambas mascot.

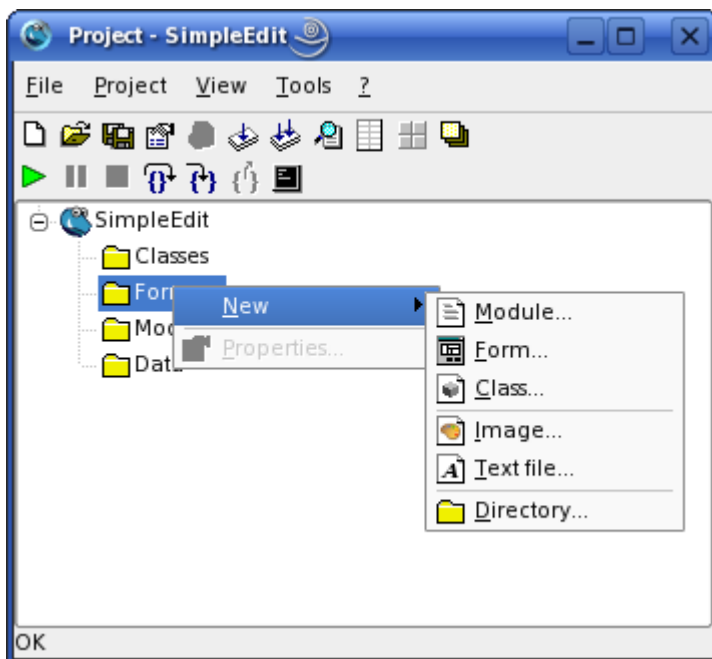
On the Tip of the day window there is a **Close** button to hide this window. Also if you do not want to see this window each time you start Gambas then there is a check box labelled **Show tips on startup** – make sure it is unchecked. If you want to see the Tip of

the day window again the select the ? (help) menu in the Gambas project manager and then the **Tip of the day** option.

The other item you might see is the Gambas mascot. Some people like this and others do not. You can select whether this is shown or not. In the Gambas project manager select the **Tools** menu and then the **Preferences...** option. This will show a dialogue where many of the settings for the Gambas user interface can be changed. Select the **Others** tab. In the Miscellaneous section there is a check box where you can select if the Mascot is shown or not. If you select not to show the Mascot it will look sad.

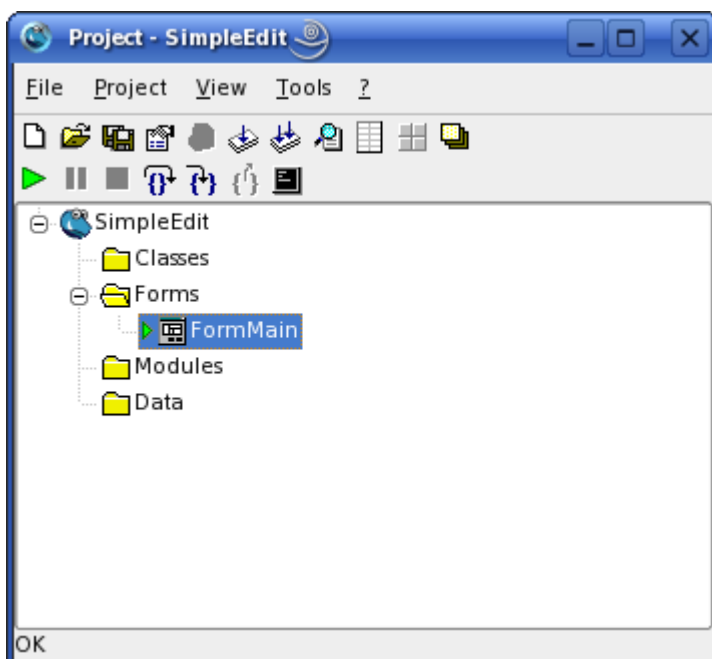
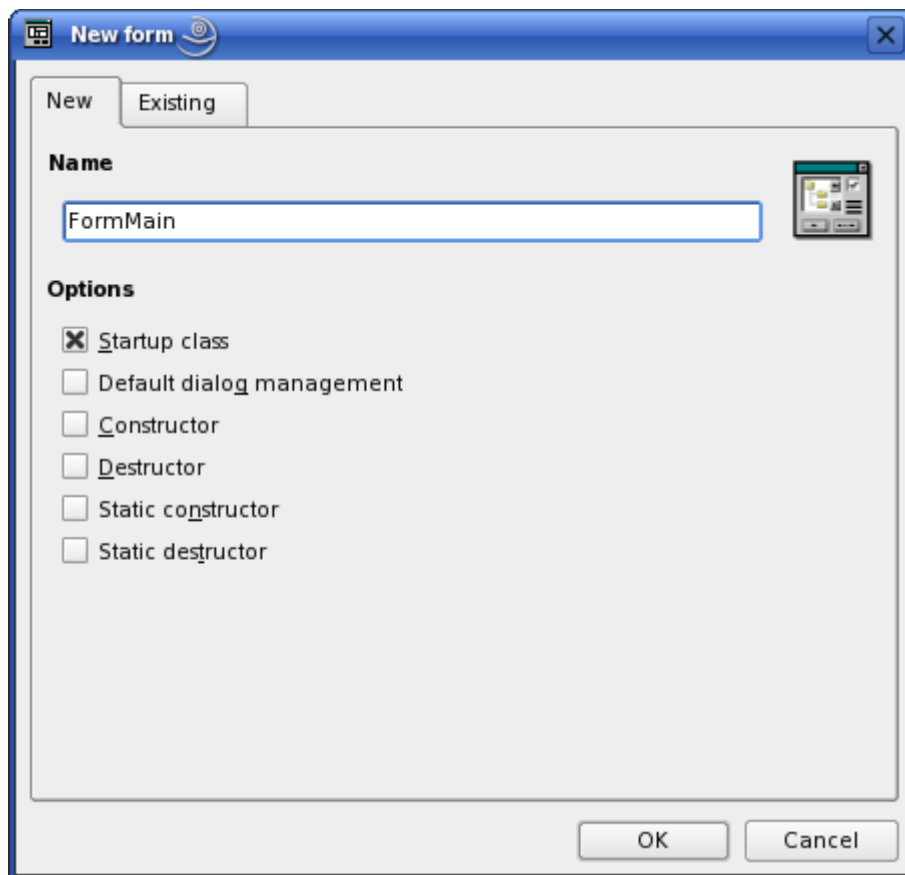
## 2.2: Creating the user interface

Now we have a Gambas project. The next step is to add some elements to the project to give us a user interface. This project will have one window. Most of this window will be taken up with a TextArea control. By using this control we get most of the functions for our text editor. The window will also have two button at the top of it. With these buttons you will be able to open and save documents from the TextArea.

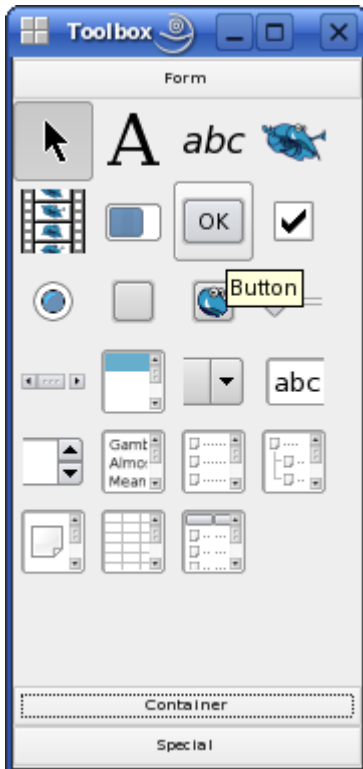


Right click in the Gambas project manager. This brings up a pop up menu. From the list select **New**. This shows a sub menu with the items we can add to our project. We need to add a Form to act as the main window to our application. So select **Form...** from this menu.

This menu option shows a dialogue where we can define some of the properties of the form we are going to create.



Give the form the name of `FormMain` and accept the other default options. The startup class is the class the Gambas runtime will first load when the application is started. There can only be one startup class in your project. We want this form to be the startup class so make sure this option is checked. Leave all the other options unchecked. Now click the **OK** button to create the form. The Gambas project manager should now look like the screen shot on the left.



We now need to add some controls on to the form we have just created. First we shall add two buttons. Make sure the form we have just created is visible. If it is not then double click on the form in the project manager. Also make sure the Toolbox is visible. If it is not then select the project manager **View** menu then **Toolbox** sub menu or press the **F6** key.

The tool box has several tabs that group the kinds of controls we can use. Make sure the **Form** tab is selected. When you move the mouse over items in the toolbox the tool tip changes to show the type of object. Select the **Button** control in the tool box.

Place two **Buttons** at the top of the form window. You can drag and drop items from the tool box onto a form designer window. It is much easier to do than it is to describe how to do it!



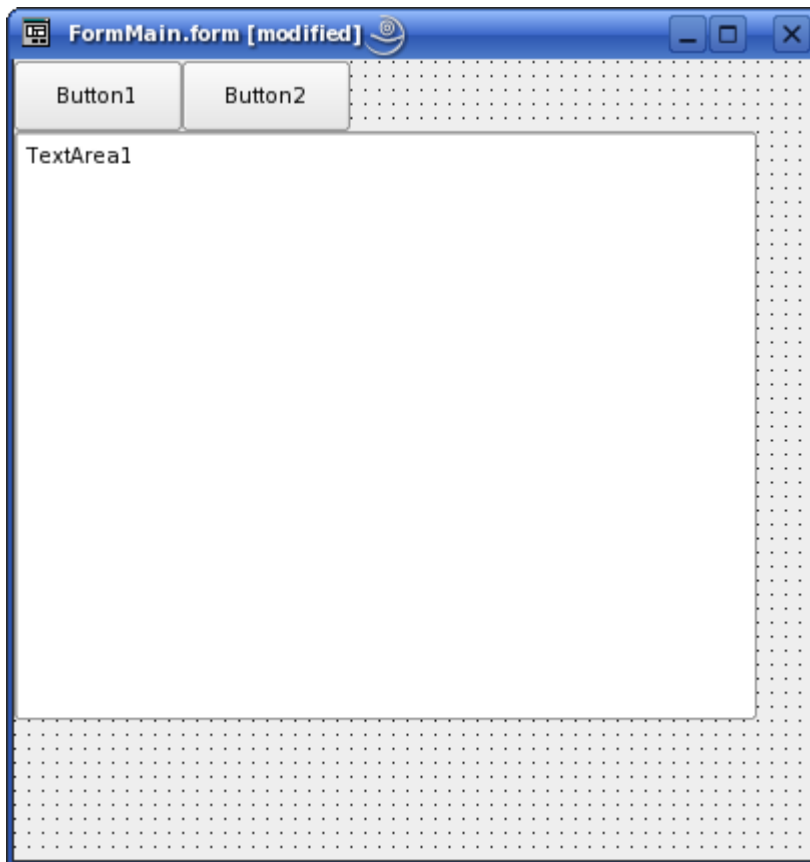
Now do the same thing with a **TextArea**. Place the **TextArea** under the **Buttons**.

Make sure the TextArea is pushed up under the two Buttons and on the far left of the form,



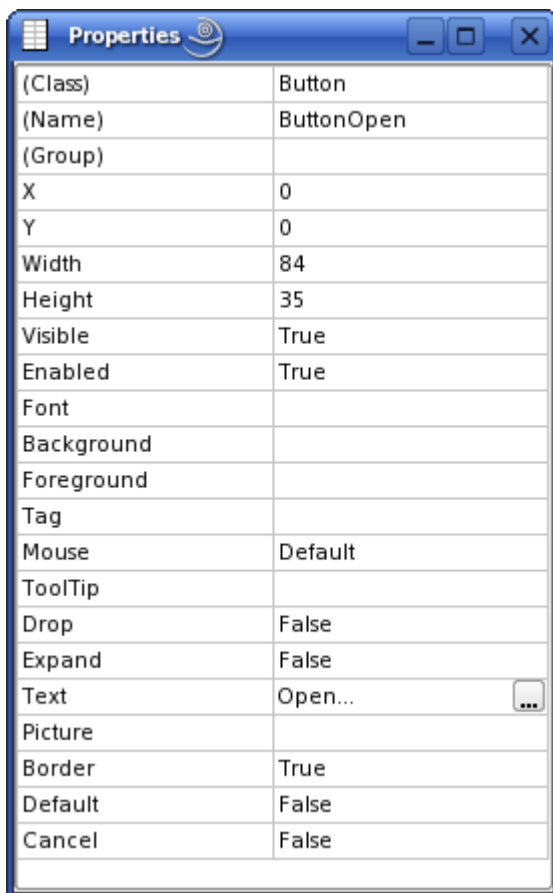
In Gambas version 1 you will see only one tab for all the controls.





Your FormMain should now look something like the screen shot of the left. The exact size of the TextArea does not matter so long as the top, left position is correct. This is because we are going to resize this TextArea at runtime.

The controls will have their default name. However these default names do not give and information about how we intend to use them. It is good programming practice to change their names to something meaningful for our application. This way we will write self-documenting code.



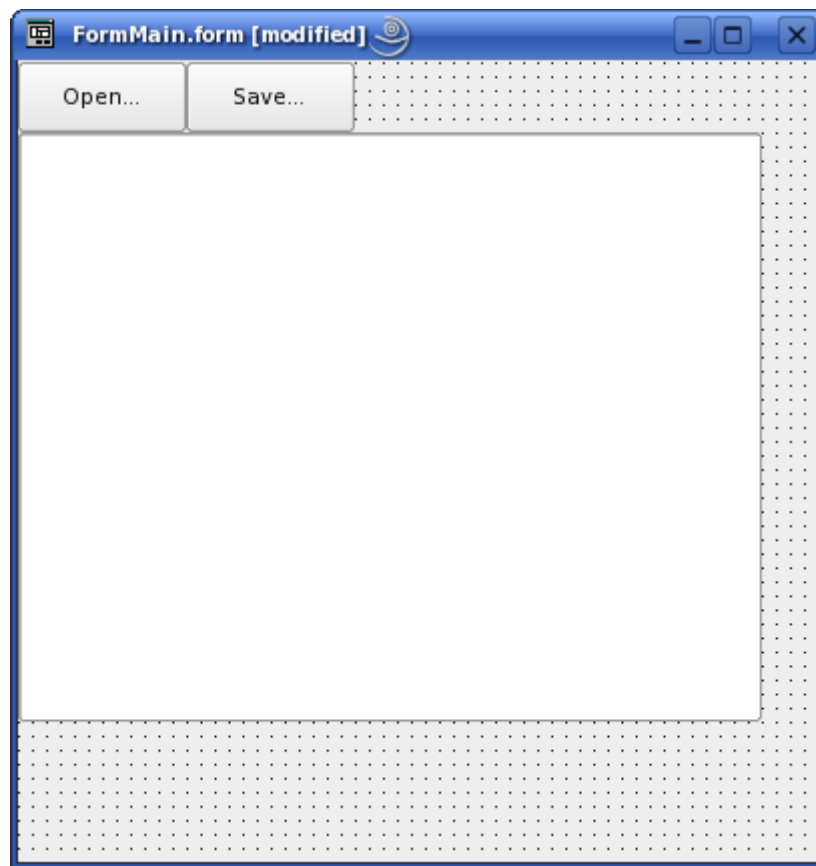
We are now going to change some of the properties of these controls. If the properties window is not visible then select the project manage **View** menu then the **Properties** sub menu or press the **F4** key. Now click on Button1 in the form designer. This will show the properties for this button. We are going to change the Name property to ButtonOpen and the Text property to Open . . .

Now we shall change the some properties for the other controls on the form. Click on Button2 in the form designer. This will show the properties for Button2. We are going to change the Name property to ButtonSave and the Text property to Save . . . Finally click on TextArea1 in the form designer. We are going to change the Name property to TextAreaEdit and then delete the text in the Text property. In other words make the Text property empty.

Finally we shall change one property on the Form. Click on the form then change the Border property to Resizable. The following table list the properties we have changed.

<i>Default Name</i>	<i>Property</i>	<i>New Value</i>
FormMain	Border	Resizable
Button1	Name	ButtonOpen
	Text	Open...
Button2	Name	ButtonSave
	Text	Save...
TextArea1	Name	TextAreaEdit
	Text	<none>

Our form should now look like the following screen shot when viewed in the form designer.



That all we need to do with the form designer. In many ways it is harder to describe what to do than do it. We can see how easy it is to build a user interface in Gambas.

### **2.3: Adding the code**

It is now time to add some code. In fact we do not need to add very much code. All we need to handle are resizing the TextArea when the form is resized. We also need to handle opening and saving a text file when the user clicks on one of the buttons.

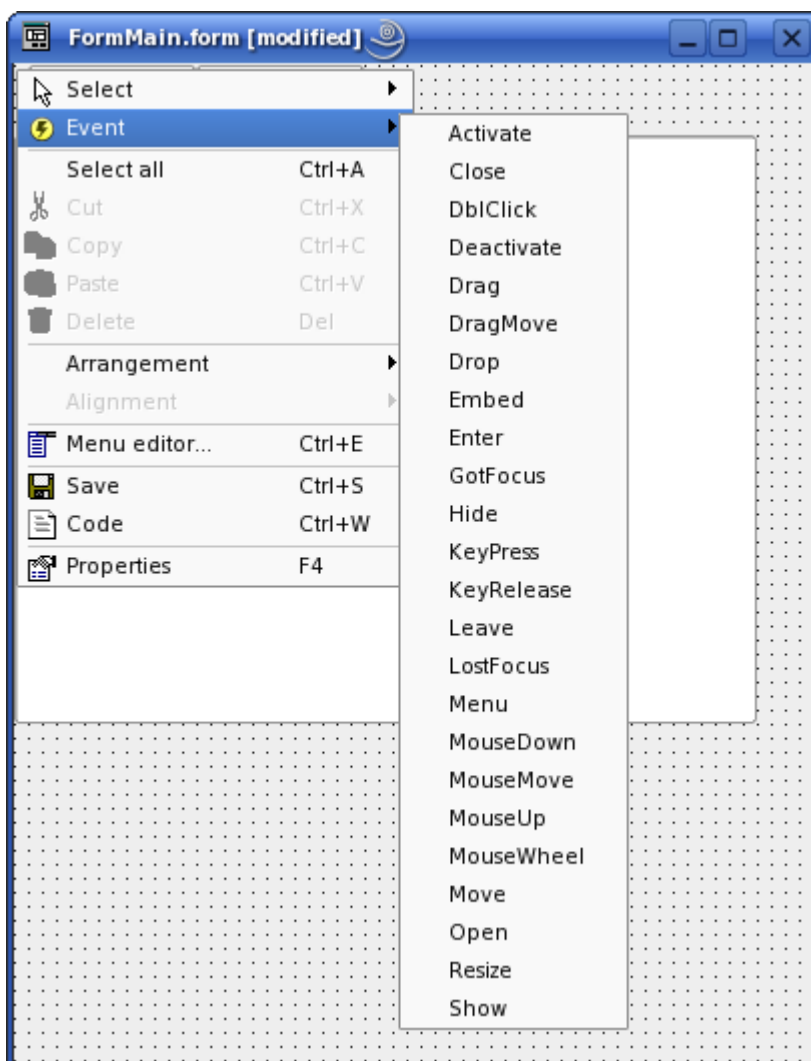
In Visual Basic you write code to handle events. Events occur when something happens to a control or object. The control or object fires the event to allow other controls or objects to respond and do something when the event happens. For a form object possible events include:

- When the form is opened
- When a key is pressed
- When the form get clicked by the user
- When the form is closed

Along with many other events. For a button possible events include when it is clicked by the user

and when it is double clicked. In Visual Basic you write code to hook up the events fired by controls or objects to the actions you want your application to perform.

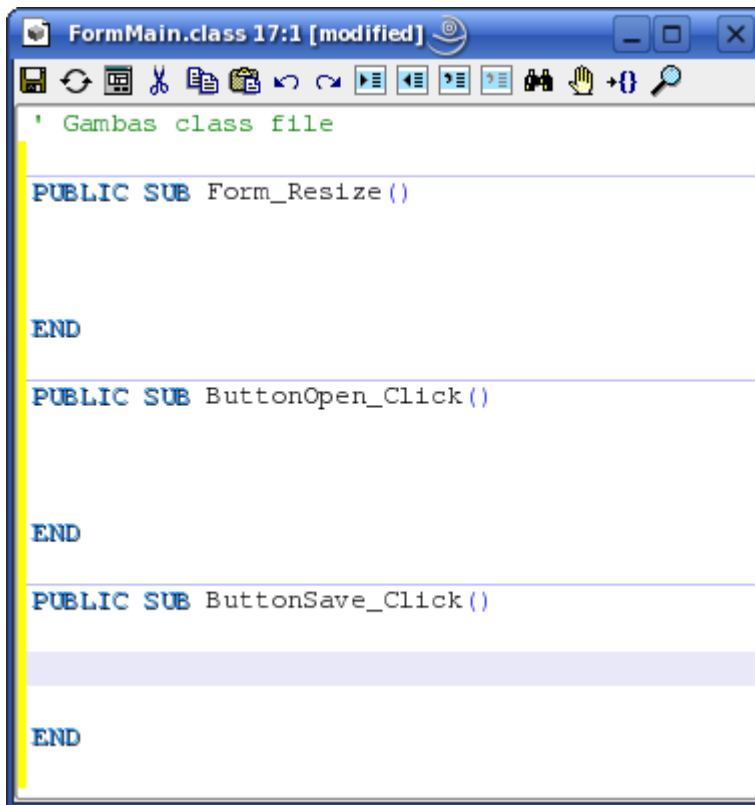
When you double click on a control or form in the Form designer it shows the default event handler for that object in the code editor. If the code does not exist for the default event handler it is created for you. For a form the default event is `Form_Open`. For this application we do not need to do anything when this event is fired. However we do need to add an event handler for the `Form_Resize` event.



To add the form resizing event handler right click on the form. (Make sure you right click on the form, not on the TextArea or one of the Buttons.) This will show a pop up menu. From this menu select **Event**. This will show a sub menu of all the events you could handle that are generated by the form. Select **Resize** from this sub menu.

This will add a stub for the `Form_Resize` event to the code in the forms class. Now we shall add the event handlers for the button click events. As the click event is the default event for the button it is much simpler to add. In the Gambas form designer Double click on the **ButtonOpen** button to add the `ButtonOpen_Click` event. Finally double click on the **ButtonSave** button to add the `ButtonSave_Click` event.

These are all the events we want to handle in this application. At the end of creating the event handles the Gambas code editor should look something like the following screen shot.



Now we shall add the code from the following listing to these event stubs. Note that ↪ icon in the following listing means that this is a continuation of the previous line. Hence for the `Form_Resize` event need only add one line of code. So enter the following code into the code editor.

FormMain.class

```
' Gambas class file

PUBLIC SUB Form_Resize()
    TextAreaEdit.Resize(ME.ClientWidth, ME.ClientHeight -
↪ TextAreaEdit.Top)
END

PUBLIC SUB ButtonOpen_Click()
    IF Dialog.OpenFile() THEN RETURN
    TextAreaEdit.Text = File.Load(Dialog.Path)
CATCH
    Message.Error(Error.Text)
END

PUBLIC SUB ButtonSave_Click()
    IF Dialog.SaveFile() THEN RETURN
```

```
File.Save(Dialog.Path, TextAreaEdit.Text)
CATCH
    Message.Error(Error.Text)
END
```

You might be wondering what some of this code does. So let's look in detail at this code. In the first procedure we resize the TextArea whenever the form is resized. This resize event also occurs when the form is first created so we do not have to add extra code to handle this case.

The first line of code tells us we are going to handle the form resize event. The name for event handlers have the following format:

- The name of the object that fires the event
- An underscore
- The name of the event

So here we are saying we are going to handle the Resize event on the Form object.

```
PUBLIC SUB Form_Resize()
```

We then call the Resize method on the TextArea. Here we tell the TextArea what size we want it to be.

```
    TextAreaEdit.Resize(ME.ClientWidth, ME.ClientHeight -  
↳ TextAreaEdit.Top)
```

The Resize method takes parameters that set the width and height of the TextArea. We calculate the size based upon the size of the current form. ME stands for the current object, in this case the current form. Note that we use ClientWidth and ClientHeight when we get the size of the current form. The ClientXXX properties always give the correct size. The form also has Width and Height properties. But the value they return depends on whether the form is maximised. So they might not return the value you expect. Also we want to leave space at the top of the form so that our buttons are visible. So we have to deduct the value for the top of the TextArea from the height of the form.

The final line for the procedure signals where the code for the event ends.

```
END
```

The next line tells us we are handling the click event for the open file button.

```
PUBLIC SUB ButtonOpen_Click()
```

The first thing we need to do in this procedure is get the path of the file the user wants to open. The Gambas Dialog object has a number of methods with which you can display a number of common dialogs. Among these is an open file dialogue. Also if the user cancels the dialogue we should exit our procedure and do nothing further. Giving the user the chance to cancel an action is good

programming practice. The `OpenFile` method returns `FALSE` when the user clicks on the **OK** button and `TRUE` otherwise. By placing this statement inside an `IF` we can check the return value of the `OpenFile` method. The `RETURN` statement causes the procedure to quit immediately. So if the user does not click on the **OK** button after selecting a file name we then we exit from this procedure. At first this line looks somewhat counter intuitive. But this way of calling the dialog does works very well when you get used to it.

```
IF Dialog.OpenFile() THEN RETURN
```

When we reach the next line we know the user has clicked on the **OK** button and selected a file. The `Dialog` object has a property called `Path` which we can use to get the full path to the selected file.

You use the `File` object when performing file input and output. The `File` object also has useful methods for quickly reading a text file. Here we use the `File.Load()` method. It takes as parameter the path to our file and then returns the content of the file as a text string. This function handles all the opening, reading and closing of the file for us. It is easy to use this function when we want the entire content of a plain text file. We pass the content of the text file `TextArea`'s `Text` property. This replaces the entire content of the `TextArea` with the content of the file we have opened.

```
TextAreaEdit.Text = File.Load(Dialog.Path)
```

We also need to handle any possible errors in our application. This is especially true when you communicate with anything outside your application. Here we have a dialogue where the user enters input. We also have a file that could contain almost anything. Without the following error handler our program would crash if the user selected an invalid file or a file where they did not have read permission. If any error occurs then the Gambas runtime jumps to this `CATCH` statement. Here we will just display a message to the user. By using the `Error` method on the `Message` object we display a error icon in the message box. Here we have kept the message as simple as possible. We simply show the text from the error that was created.

```
CATCH
```

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

```
END
```

Code after a `CATCH` statement is not executed when no error occurs. So with valid files the user will never see the message box.

The code for the save file button is very similar to that of the open file button. The first line tells us we are handling the click event for the save button.

```
PUBLIC SUB ButtonSave_Click()
```

This time we call the save file dialog which is very similar to the open file dialog we used above. But this dialog has functionality more appropriate to saving a file. Again this dialog returns `FALSE` when the user clicks on the **OK** button and `TRUE` otherwise. So if the user does not click on the **OK** button after selecting a file name we exit from the procedure.

```
IF Dialog.SaveFile() THEN RETURN
```



Again we use the File object for saving. This object also has a useful method for saving text files. We simply need to tell it what path to use when saving the file and what string to save. Here, of course, we are going to save the content of our TextArea.

```
File.Save(Dialog.Path, TextAreaEdit.Text)
```

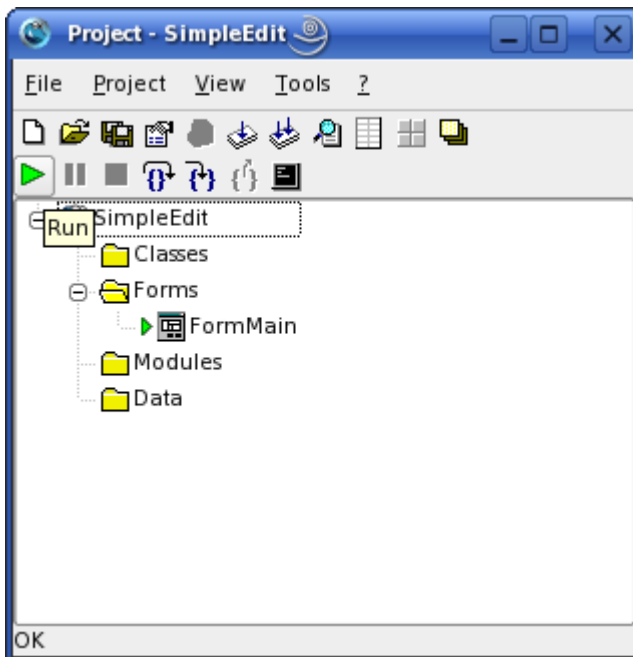
Here again we have the same kind of error handler as for the open file button.

CATCH

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

END

## 2.4: Running the project

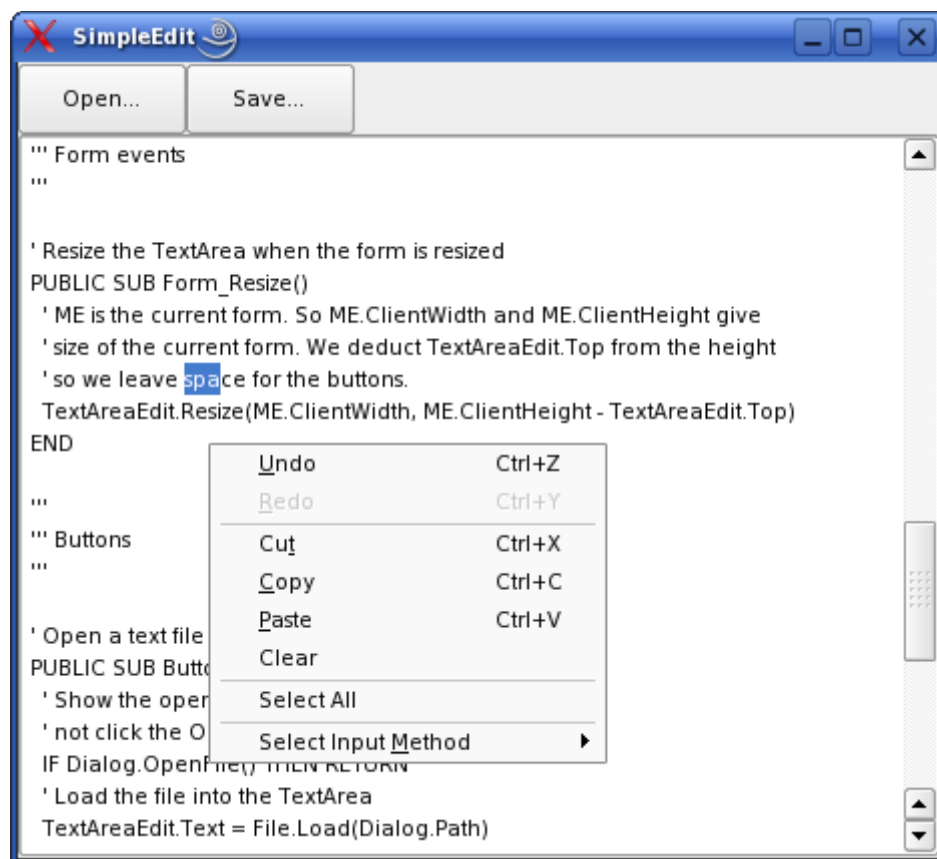


That's everything we need for this project. So let's run it. You run the project by clicking on the green **Run** button in the project manager window or by pressing the **F5** key.



If the project does not run the most likely cause is a misspelling in the code we entered. Look at the line that Gambas highlights when finding an error and check the names match the names of the controls on the form. Also check the syntax of the highlighted line matches the code example.

Here is a screen shot of the final running application after we have opened the code file for this project. As you can see the code is the same as above, but there are extra comments in this file. These comments are in the examples that come with this tutorial. The screen shot also shows a pop up menu with options for editing the text. This menu is part of the TextArea control. You get this menu for free by using this control. Try right clicking on the TextArea in your application when it is running.



When you develop any kind of application you need to test its functionality. Above all you should not be able to crash the application. It should be able to handle any errors gracefully and give the user a useful error message. Even for an application as small as this you should have a test plan.

But what should you test? These are the most likely problem areas with any application:

- **Your code.** Your tests should cover every line of code in the application. This includes all branches is a selection. Also every loop should be tested for zero, one and many iterations.
- **Interaction between objects.** It is often in the interaction between objects where errors occur. There should be a test for every achievable interaction between objects (interaction that are only theoretical need not be tested).
- **Interfaces to the world outside you application.** Any useful application has to communicate with something outside of itself. Your application should not trust any data from outside of itself until it has been validated inside the application. This includes user input as well and communications to any devices. You also need to test for any mix of otherwise valid inputs that together are invalid. Also any invalid data should be handled appropriately.

Normally you would not test the following items (except when building a critical system — such as application for use in a hospital):

- Operating system functions. These could be system calls or shell commands.

- In-built controls for your programming environment.
- Any third party components in your application.

These are items you have less control over. If there are faults with any of these items then you would properly have to change your application to work around the fault.

A good way to test your application is to produce a table of tests. Ideally you would produce this table before you started coding the application. If new features are later added to the application then add more tests to the table. In the table you would write down each of the test and the result you expect for the test to pass. You need to state the expected result so there is no ambiguity if your application does something different. The final column is for you to record the results of the testing process. Our first set of tests deal with resizing the application window.

<i>Form Resizing Tests</i>		
<i>Test</i>	<i>Expected Result</i>	<i>Pass/Fail</i>
Resize the window using right edge	Window should be resize and the text area resized to fit in the window.	
Resize the window using bottom edge	As above.	
Resize the window using left edge	As above.	
Resize the window using top edge	As above.	
Resize the window using north/west corner	As above.	
Resize the window using south/west corner	As above.	
Resize the window using south/east corner	As above.	
Resize the window using south/east corner	As above.	
Minimize the window and restore it	The window should be correctly resize when restored.	
Maximise the window and restore it	Window should resize and the text area resize to fit in the maximised window. The window should be correctly resize when restored.	
Make the window height smaller then the top row of buttons	You will not see the text area. No error should result.	
Make the window width smaller than the width of both buttons	No error should result.	

The next set of tests are for the button which opens a text file.

<b>Open File Tests</b>		
<b>Test</b>	<b>Expected Result</b>	<b>Pass/Fail</b>
Open text file	File displayed in text area	
Open an empty text file	Text area empty	
Open a very large text file	File displayed in text area.	
Open a binary file	File displayed in text area. Content is properly not very meaningful	
Try to open a file that does not exist	Error message	
Try to open a file where you do not have read permission	Error message	

The final set of tests are for the button which saves a text file. When performing the save file tests *make sure you do not overwrite any files you need or any system files.*

<b>Save File Tests</b>		
<b>Test</b>	<b>Expected Result</b>	<b>Pass/Fail</b>
Enter some text in the text area and save a <i>new</i> text file to your Home directory	File saved	
Enter some text in the text area and save to an <i>existing</i> file in you Home directory.	File saved	
With no text in the text area save an empty file to your Home directory	File saved	
Open an binary file and save this as a new file	File saved	
Try to save the file to a directory where you do not have write permission	Error message	
Try to save the file to a directory that does not exist	Error message	

We have ended up with more tests than lines of code in our application. This may seem overkill for your first project. But the more thorough you are at testing the more useful your applications will be to other users. Getting used to a tough testing regime from the start will pay dividends in the long run. The real problem is being as meticulous as you can in your tests. You need to cover any many options as possible and make sure all code is covered. One hundred percent coverage is not possible, but you should aim for this.

So work through the list of text and make sure you record if they pass or fail. If any test fails you will need to debug the code to find the fault. Also are there any tests that should be added to the

list?



One of the best pieces of advice I can give you is:

***Use the applications you build yourself.***

If you build a paint program then use it for *all* your drawing tasks. That way you find what features you need to add. (And also what features seemed a good idea at development time, but are little used.) Also after a few weeks make sure you think again about how easy it is for a new user to learn your application.

That was not too hard. We have created a basic but perfectly functional text editor. We have achieved this by using the power of existing components. The core functionality for our text editor was already contained in the TextArea control.

You could go on adding features to this project and develop it as your own text editor. For example you could add some buttons to cut, copy and paste text. Then hook these up to the methods on the TextArea control to handle this functionality.

---

### 3: Drawing Project: ImageShow

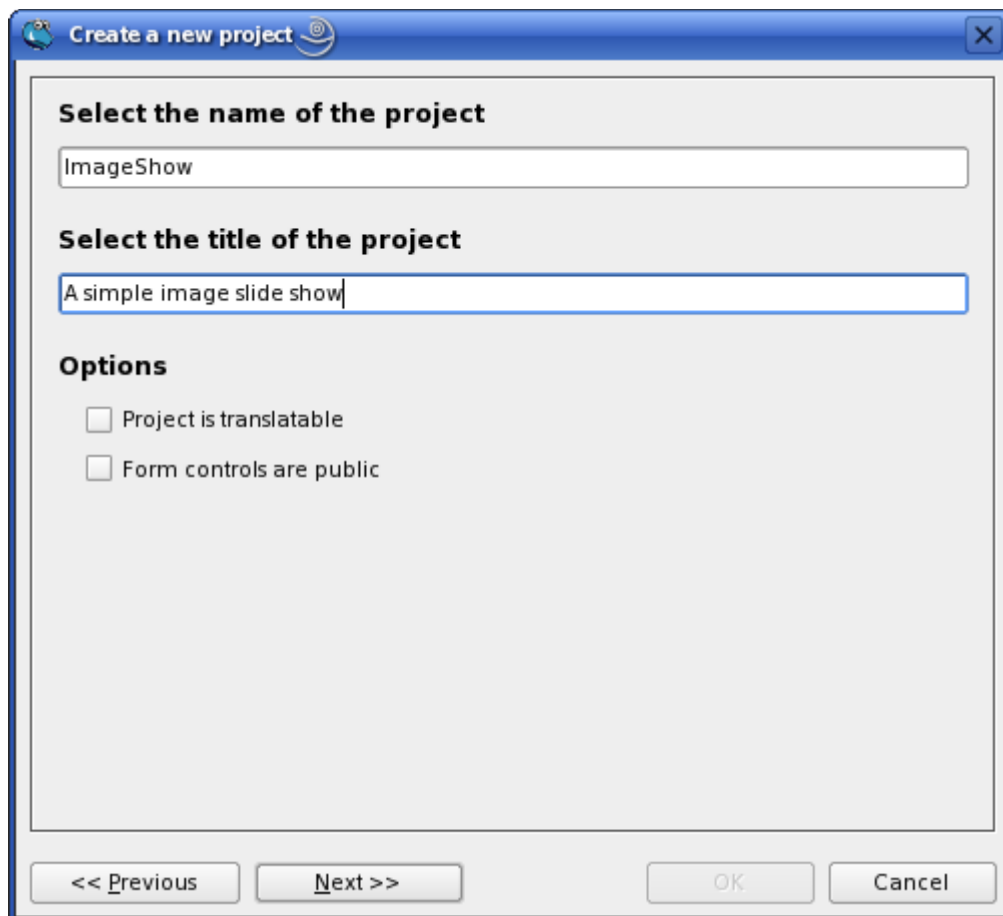
With this project we are going to create a simple slide show application. We want the project to have the following functionality:

- The application should display images from a selected directory.
- The display should be full screen.
- Images should be displayed full size and centred on the screen.
- If an image is larger then the screen then it should be scaled to fit on the screen keeping the aspect ratio of the image.
- Once a directory is selected the application will scroll through the images displaying one every 15 seconds.
- Pressing the Escape key will quit the application.
- Pressing F1 will display a brief help screen.
- Pressing space bar or right arrow will scroll to the next image.
- Pressing the backspace key or left arrow will display the previous image.
- Pressing the 'D' key will allow the user to change the image directory.
- Pressing the 'S' will start or pause the slide show
- Pressing the 'T' key will display information about the image/show.
- The mouse should be hidden when images are displayed.

This looks like a long list but as we shall see the final application is not that complex.

#### 3.1: *Creating the project*

The first step is to create a new Gambas project. So open Gambas and select **New project...** This will start the new project wizard. The first page of the wizard simply shows a welcome screen with details of how to use the new project wizard. Click the **Next >>** button to for the next page of the wizard. This shows a page were we can select the type of Gambas project we want to build. We are going to use the default option of a graphical project. So click the **Next >>** button. This shows a page when you enter the name and title for your project we are going to create.



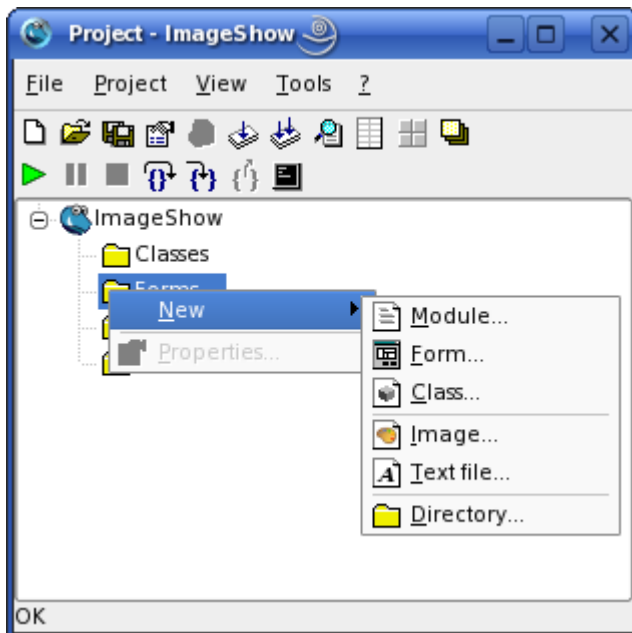
Enter the name of `ImageShow` and the title of `A simple image slide show`. Leave the other options unchecked. Click on the **Next >>** button. This shows a page where you select the directory where you want to save the project files.

Select the location where you want to save the project. Then click the **Next >>** button. This final page lists the options you have selected for the project. Check through the options and then click the **OK** button. We have created a new Visual Basic project and Gambas will open showing the Gambas project manager.

### 3.2: *Creating the user interface*

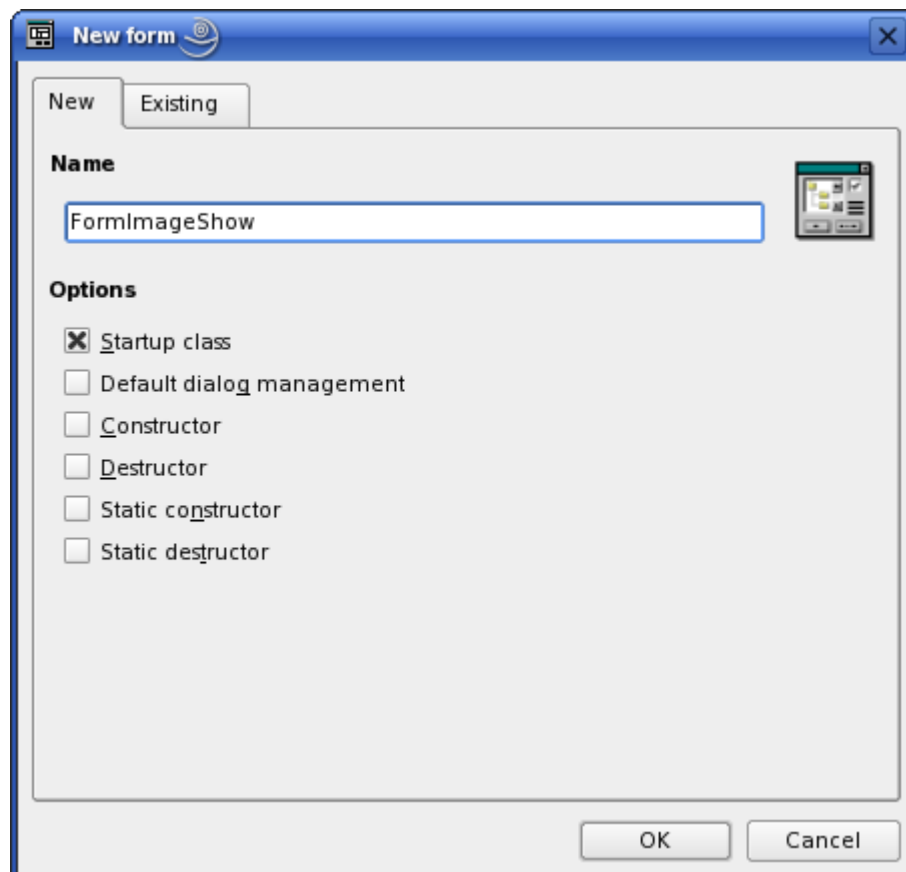
Now we have a Gambas project. The next step is to add some elements to the project to give us a user interface. This project will have one window that will be displayed full screen. All of this window will be taken up with a `DrawingArea` control. We could draw the image directly on the form but using a `DrawingArea` gives us some extra functionality. We will also put a timer on the form. This is so we can fire events to change images as set intervals.



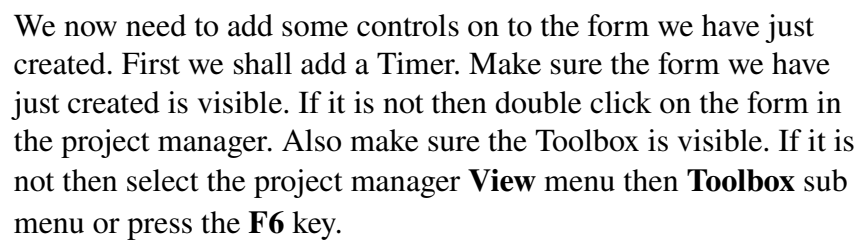


Right click in the Gambas project manager. This brings up a pop up menu. From the list select **New**. This shows a sub menu with the items we can add to our project. We need to add a Form to act as the main window to our application. So select **Form...** from this menu.

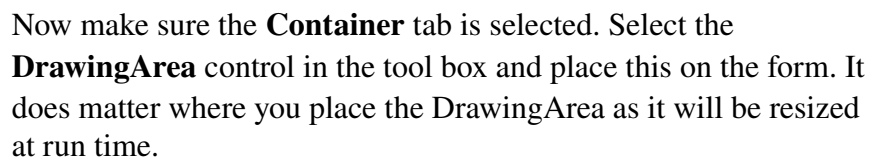
This menu option shows a dialogue where we can define some of the properties of the form we are going to create.



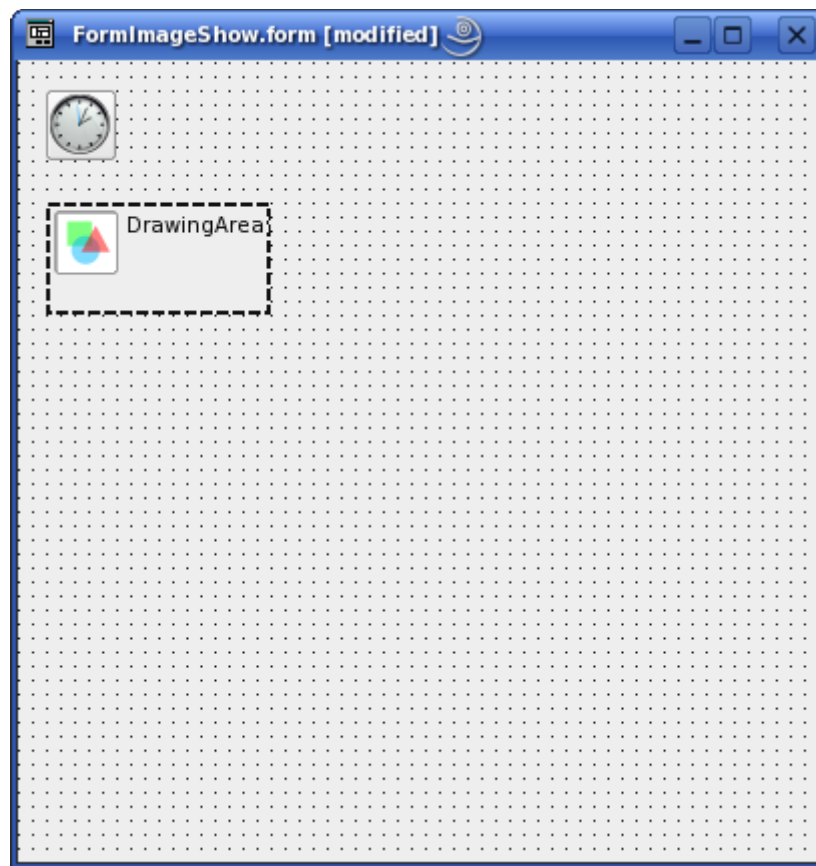
Give the form the name of `FormImageShow` and accept the other default options. The startup class is the class the Gambas runtime will first load when the application is started. There can only be one startup class in your project. We want this form to be the startup class so make sure this option is checked. Now click the **OK** button to create the form.



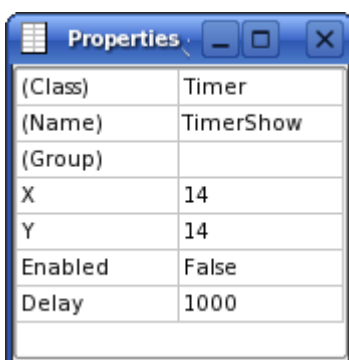
Place a **Timer** on the form window. You can drag and drop items from the tool box onto a form designer window. It does not matter where you put the timer as it will not be visible when the program is run.



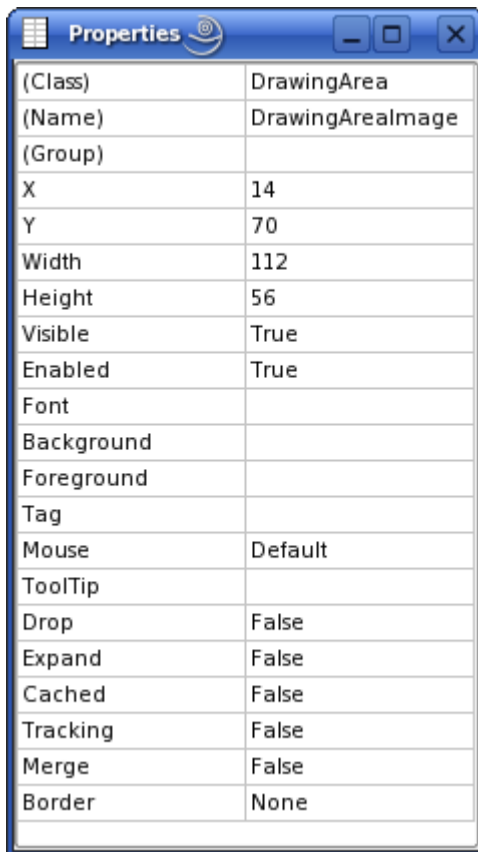
At this point our FormImageShow should look like the following screen shot.



The controls will have their default name. However these default names do not give any information about how we intend to use them. It is good programming practice to change the names to something meaningful for our application. This way we will write self-documenting code.



We are now going to change some of the properties of these controls. If the properties window is not visible then select the project manage **View** menu then **Properties** sub menu or press the **F4** key. Now click on Timer1 in the form designer. This will show the properties for this object. We are going to change the Name property to TimerShow.



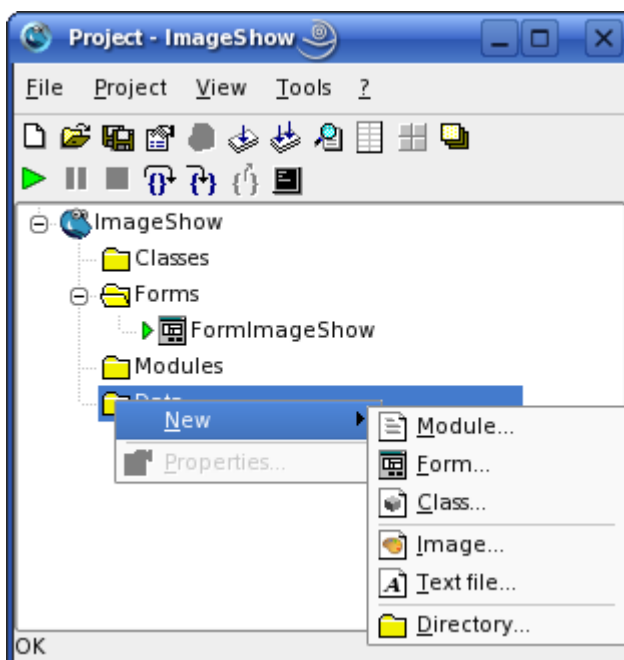
Click on the DrawingArea object and change the Name property to DrawingAreaImage.

That all of the properties we are going to change in the Form designer. Some other properties of these objects will be changed in the code.

We want to show some help informations to the user when they press the F1 key. The method we are going to use is suitable when the required amount of help is small and will fit on one page. The Gambas Message box is quite a flexible class and will take a HTML formatted message. So with the single line of code:

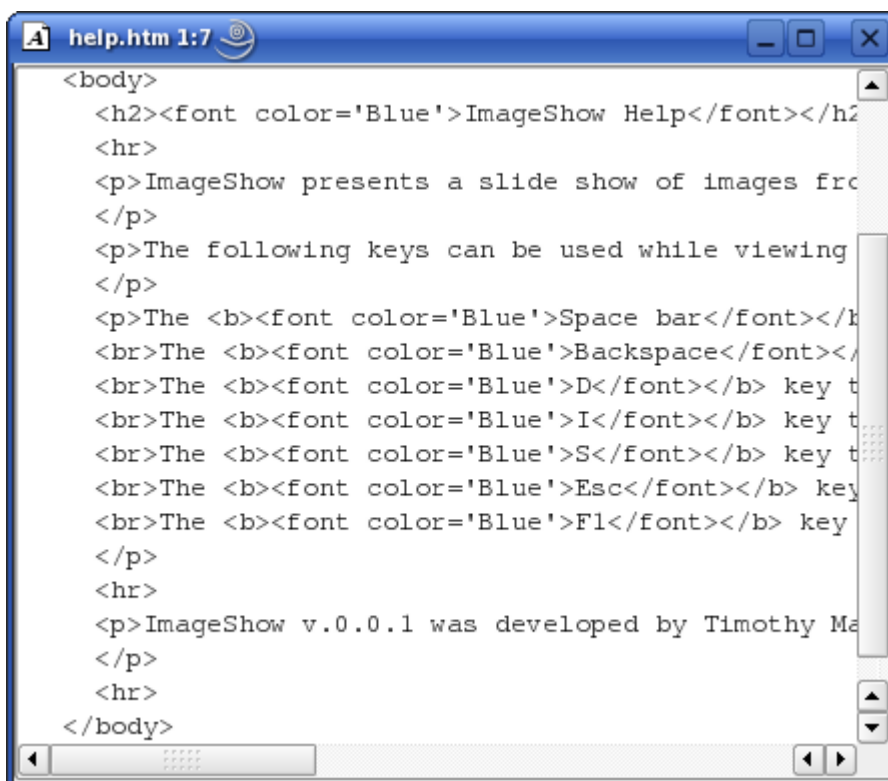
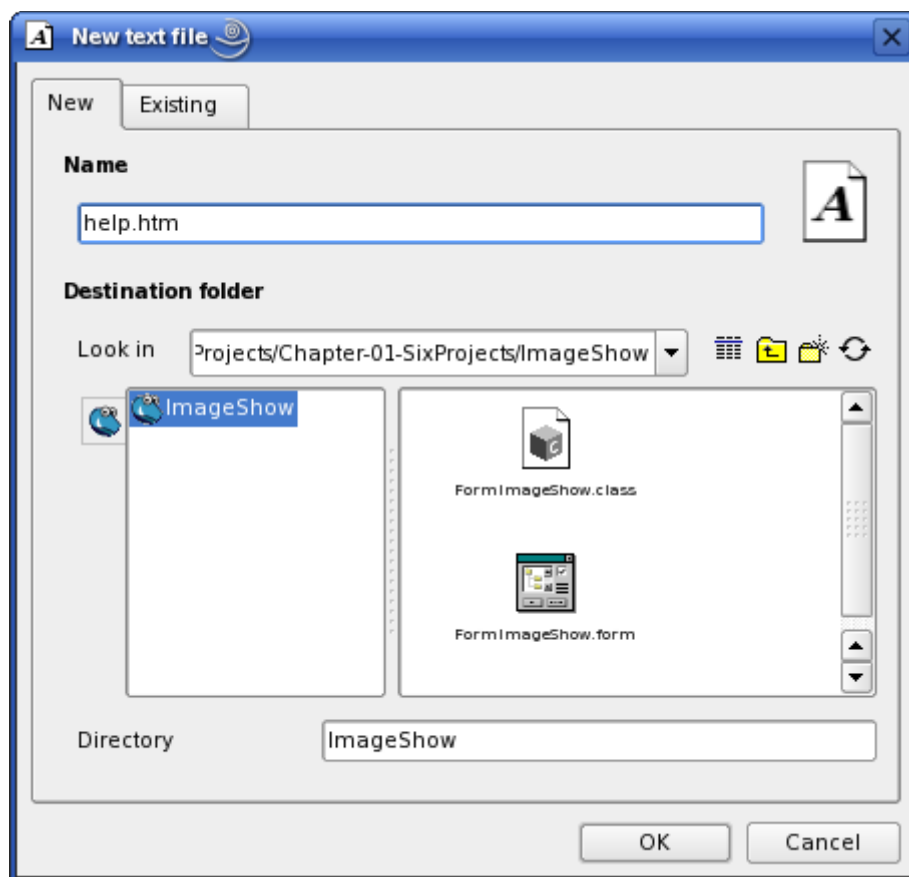
```
Message.Info(File.Load("help.htm"))
```

we can display a HTML file.



So let create the HTML file in Gambas. Right click in the Gambas project manager. This brings up a pop up menu. From the list select **New**. This shows a sub menu with the items we can add to our project. We need to add a Text file so select **Text file...** from this menu.

This shows a dialogue where we give the text file a name.



Give the file the name `help.htm` and press the **OK** button to save it. This will add the file to the data section of your project and will open the new text file in the Gambas text editor window. The screen shot on the left shows the Gambas text editor window after we have entered some HTML into it.

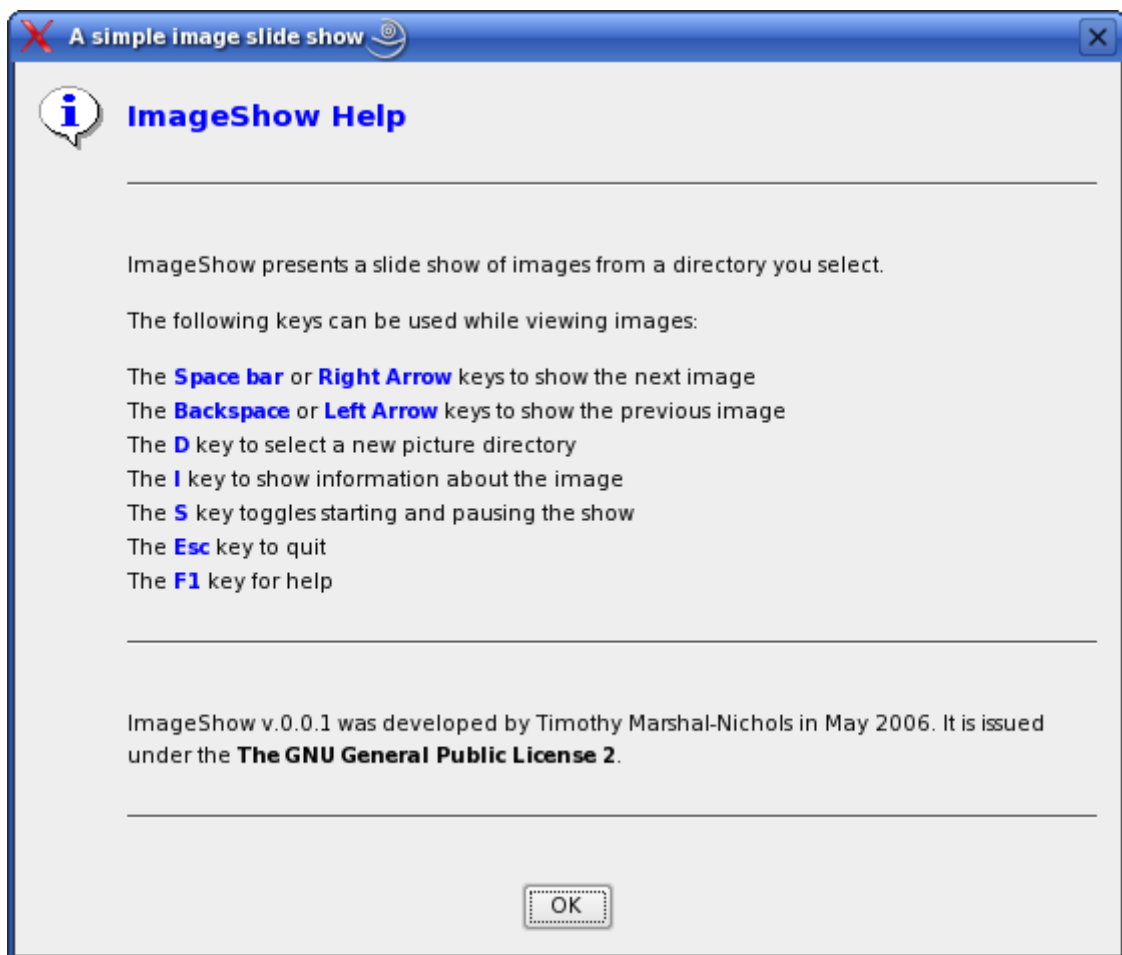
This is not a tutorial about HTML – there are plenty of those on the internet. So we are not going to say much about this file. Just enter the following HTML into the text editor. Note that, as before, the ↵ icon in the following listing means this line is a continuation of the previous line.

help.htm

```
<html>
  <head>
    <title>ImageShow Help</title>
  </head>
  <body>
    <h2><font color='Blue'>ImageShow Help</font></h2>
    <hr>
    <p>ImageShow presents a slide show of images from a directory
    ↵ you select.
    </p>
    <p>The following keys can be used while viewing images:
    </p>
    <p>The <b><font color='Blue'>Space bar</font></b> or <b><font
    ↵ color='Blue'>Right Arrow</font></b> keys to show the next image
    <br>The <b><font color='Blue'>Backspace</font></b> or <b><font
    ↵ color='Blue'>Left Arrow</font></b> keys to show the previous
    ↵ image
    <br>The <b><font color='Blue'>D</font></b> key to select a new
    ↵ picture directory
    <br>The <b><font color='Blue'>I</font></b> key to show
    ↵ information about the image
    <br>The <b><font color='Blue'>S</font></b> key toggles
    ↵ starting and pausing the show
    <br>The <b><font color='Blue'>Esc</font></b> key to quit
    <br>The <b><font color='Blue'>F1</font></b> key for help
    </p>
    <hr>
    <p>ImageShow v.0.0.1 was developed by Timothy Marshall-Nichols
    ↵ in May 2006. It is issued under the <b>The GNU General Public
    ↵ License 2</b>.
```

```
</p>  
<hr>  
</body>  
</html>
```

The Gambas controls that accept HTML formatting only take a subset of HTML tags. So it is sometimes trial and error to find out what works. The following screen shot shows what this HTML looks like in a message box when the application is run and the user presses the F1 key.



That all of the elements we need for the user interface. All the rest of the applications functionality is going to be done in code.

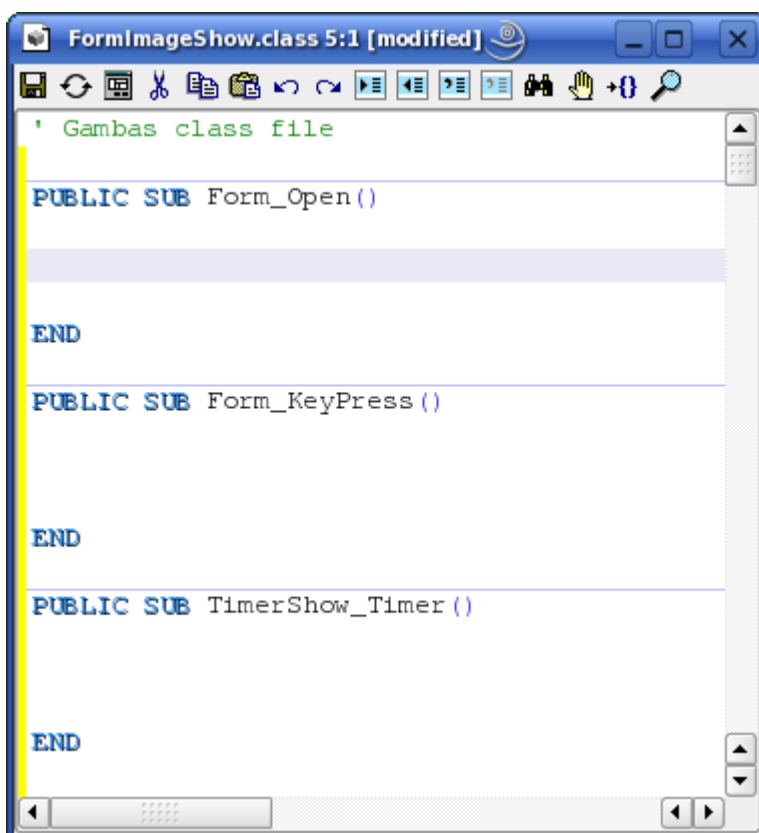


### 3.3: Adding the Code

There are three event we need to handle in this application. We need some set-up code when the application opens. When the user presses a key on the keyboard we need to check the key press and, if required, respond. We also need to update the screen image every 15 seconds when the slide show is running.

So lets add these events to our application. The first event is easy double click on the FormImageShow in the form designer. This will add the event `Form_Open` to the `FormImageShow` class.

Next we shall add the event for a user key press. Right click on the on the `FormImageShow` in the form designer. This will show a pop up menu. Select the **Events** option and then from the sub menu select **KeyPress**. This will add the event `Form_KeyPress` to the `FormImageShow` class. Finally double click on the `TimerShow` object on the `FormImageShow`. This will add the event `TimerShow_Timer` event to `FormImageShow` class.



When you have added these events the `FormImageShow` class should look like this screen shot.

Now we shall add code to the `FormImageShow` class. In this tutorial chapter we shall split the code into chunks so that we can describe each section. We need two variables to store information about

our slide show. The first variable is a array of strings called `pictureFiles`. Here we shall store the file paths to all in images in our slide show. The next variable is `pictureNumber` which will hold the index number of the currently displayed image.

```
FormImageShow.class
```

```
PRIVATE pictureFiles AS NEW String[]  
PRIVATE pictureNumber AS Integer
```

### [Continued Below](#)

Notice that we declare these variables as `PRIVATE`. Later on you will see that variables declared inside a procedure or function are declared using the `DIM` statement. However in Gambas when declaring variables at module or class level you need to use the `PUBLIC` or `PRIVATE` keywords.

We shall add code to the `Form_Open` event we created earlier. This event fires when the form is first shown and only fires once. It is a useful place to put any set-up code you need for a form. We start by setting the forms background to black and making the form full screen. (Gambas version 1 users see the [note below](#).)

We then set the changes we need to the `DrawingArea`. We resize the `DrawingArea` with the `move` statement. Normally we would use the `ME.ClientWidth` and `ME.ClientHeight` properties to get the size of the current form. In most cases these are the properties you should use. But here they are still set to the old form size. They do not appear to become set to the values we want until the form resize event is called. As we want the form to be full screen we can use the `Desktop` object to get the values we need. The `Desktop` object gives us some useful information about the screen. Also for the `DrawingArea` we set the `Cashed` property to `TRUE`. When this property is set the `DrawingArea` will automatically handle refreshing of itself if the current picture needs to be redrawn. This means we let the `DrawingArea` redraw the current picture when needed. We only need to think about drawing when we want to change the picture.

We then call two functions. The first allows the user to select a directory from which to build a list of images. The next function will display an image. The code for these functions is described [later on](#). After calling these functions we hide the mouse so it does not detract from the image.

Finally we set up the show timer. We set the delay time between images to 15 seconds. But we only start the show timer if we have some images.

```
FormImageShow.class - Continued
```

```
PUBLIC SUB Form_Open()  
  ' Set form properties  
  ME.BackColor = Color.Black  
  ME.FullScreen = TRUE  
  ' Set drawing area properties  
  DrawingAreaImage.Move(0, 0, Desktop.Width, Desktop.Height)  
  DrawingAreaImage.Cached = TRUE  
  ' Select image directory  
  Dialog.Path = User.Home  
  SelectImages()  
  NextPicture()  
  ' Hide mouse  
  DrawingAreaImage.Mouse = Mouse.Blank  
  ' Start show timer  
  TimerShow.Delay = 15000 ' 15 seconds  
  TimerShow.Enabled = (pictureFiles.Count > 0)  
END
```

[Continued Below](#)



In Gambas version 1 you use a slightly different method of displays the form full screen. The `ME.FullScreen = TRUE` line in the above listing needs to be changed to the following:

```
ME.State = Window.Fullscreen
```

Also to select the users home directory you use the `System` object.

```
Dialog.Path = System.Home
```

This next event handles any key press our application receives. When ever the user presses a key we select the option the user wants. We use the `TimerShow.Enabled` property as a flag to indicate if the image show is running or paused. For some of the key press options we want to restore the image show to its previous state. So we store the current state in the variable `timerState`.

The `Key` object gives us information about the last key press. The `Key.Code` property gives us the

keyboard code and we can compare this with constants in the `Key` object in our `SELECT` statement.

When the **Escape** key is pressed we use `ME.Close()` to end the application. If the **F1** key is pressed we hold the image show and then display our HTML help file in a message box. When the user closes the message box the image show state is set to the saved state.



In Gambas version 1 the message box gets lost behind the form when the form is full screen. So amend the code for the **F1** button to the following:

```
TimerShow.Enabled = FALSE
DrawingAreaImage.Mouse = Mouse.Default
ME.State = Window.Normal
ME.Border = Window.Resizable
Message.Info(File.Load("help.htm"))
ME.State = Window.FullScreen
DrawingAreaImage.Mouse = Mouse.Blank
TimerShow.Enabled = timerState
```

The help message does not look as good as in version 2 of Gambas. But at least it does not get lost.

The next two cases handle the user manually advancing the current image. When the user presses the **Right Arrow** or **Backspace** key we decrement the current image counter and then call our procedure to draw the image. Similarly if the user presses the **Left Arrow** or **Space bar** key we increment the current image counter and then call our procedure to draw the image.

When the **D** key is pressed we handle the user changing the selected directory for images. First we pause the slide show. Then we show the mouse by setting it to its default cursor. It could be confusing to the user to keep the mouse hidden. We then call two functions that we used in the [Form Open](#) event. The first allows the user to [select](#) a directory from which to build a list of images. The next function will [draw](#) an image. We hide the mouse again and start the show timer if we have some images.

The **S** key toggles pausing the slide show or starting it running again. When pausing the slide show we simply disable the slide show timer. We only allow the slide show to start if there are some images to display. If there are some images we increment the current image counter and display the next image. This gives a visual confirmation to the user that the slide show has started running again.

When the **I** is pressed we call a procedure that draws some information on the screen about the current state of the slide show and the current image. This procedure is [described later](#).

#### FormImageShow.class - Continued

```
PUBLIC SUB Form_Press()  
  DIM timerState AS Boolean  
  timerState = TimerShow.Enabled  
  SELECT CASE Key.Code  
    CASE Key.Esc  
      ' Close the slide show  
      ME.Close()  
    CASE Key.F1  
      ' Show HTML help  
      TimerShow.Enabled = FALSE  
      DrawingAreaImage.Mouse = Mouse.Default  
      Message.Info(File.Load("help.htm"))  
      DrawingAreaImage.Mouse = Mouse.Blank  
      TimerShow.Enabled = timerState  
    CASE Key.BackSpace, Key.Left  
      ' Show previous picture  
      TimerShow.Enabled = FALSE  
      DEC pictureNumber  
      NextPicture()  
      TimerShow.Enabled = timerState  
    CASE Key.Space, Key.Right  
      ' Show next picture  
      TimerShow.Enabled = FALSE  
      INC pictureNumber  
      NextPicture()  
      TimerShow.Enabled = timerState  
    CASE Key["D"]  
      ' Get a new slide show from a directory  
      TimerShow.Enabled = FALSE  
      DrawingAreaImage.Mouse = Mouse.Default  
      SelectImages()
```

```
NextPicture()
DrawingAreaImage.Mouse = Mouse.Blank
TimerShow.Enabled = (pictureFiles.Count > 0)
CASE Key["S"]
  IF TimerShow.Enabled THEN
    ' If the show is running then stop it
    TimerShow.Enabled = FALSE
  ELSE IF pictureFiles.Count > 0 THEN
    ' If the show is not running and we have pictures
    ' then start the show with the next picture
    INC pictureNumber
    NextPicture()
    TimerShow.Enabled = TRUE
  END IF
CASE Key["I"]
  ' Show some information about the show
  ShowMessage()
DEFAULT
  ' Nothing
END SELECT
END
```

### [Continued Below](#)

When the image show is running a timer event will be fired. Here we increment the counter we have for the image number by 1 and then call our function to [draw](#) the image. We wrap the drawing call with Enabling and Disabling a timer. This is properly not needed and is a hang over from coding in another Visual Basic. I have put this in so we reset the timer after having completed drawing the image.

### FormImageShow.class - Continued

```
PUBLIC SUB TimerShow_Timer()
  TimerShow.Enabled = FALSE
  ' Go to the next image
  INC pictureNumber
```

```
' Draw the next image
NextPicture()
TimerShow.Enabled = TRUE
END
```

### [Continued Below](#)

We have called this next procedure several times above. It allows the user to select a new image directory and loads the paths of the image files into `pictureFiles` array. First we call a standard dialogue that allows the user to select a directory. Calling this dialogue is similar the way we opened and saved files in the [SimpleEdit](#) project above. If the user does not select a the **OK** button and a directory we return from the procedure. The `Dialog.Path` property returns the name and path of the selected directory. If the user selects a directory we clear the old list of image files.

Using the `Dir` function we can obtain a list of all files in a directory. As this list is not sorted we use the `Sort()` method on a string array to sort the file list. We then use a `FOR NEXT` construct to loop through each of the files. The `File` class provides some useful method for extracting elements from a file path. Here we want to test the file extension for each file so we use the `File.Ext` method. In order to provide a case insensitive search for images we convert the extension to lower case. We then test the file extension for the image file types supported by Gambas. If we have found a valid image file name then we add the full file name and path to our image file list.

When we have completed the loop we check to see if we have found any image files in the directory. If we have not we display a message to the user. We then set the current image index to the first image. We have a `CATCH` here because we are interfacing with the file system and there is the possibility of file I/O errors. Hopefully the user will never see this error message.

We have built a list of images based upon the extension of the file. In most cases this will produce a valid list. However just because a file has an extension of **jpg** does not mean it is a valid **jpg** image file. A user can give a file any name they want. Hence when we display the images we shall still need some error checking code.

### FormImageShow.class - Continued

```
PRIVATE SUB SelectImages()
    DIM fileName AS String
    DIM fileList AS String[]
    DIM fileExtension AS String
    ' Get the directory name from the use
```

```
IF Dialog.SelectDirectory() THEN RETURN
pictureFiles.Clear()
fileList = Dir(Dialog.Path)
fileList.Sort()
FOR EACH fileName IN fileList
    fileExtension = Lower(File.Ext(fileName))
    ' Only select image files
    IF fileExtension = "png" OR fileExtension = "jpeg" OR
        fileExtension = "jpg" OR fileExtension = "bmp" OR
➔ fileExtension = "gif" OR
        fileExtension = "xpm" THEN
        ' This is a image so add to image list
        pictureFiles.Add(Dialog.Path & "/" & fileName)
    END IF
NEXT
IF pictureFiles.Count = 0 THEN
    Message.Info("No images found in the directory:\n\n" &
➔ Dialog.Path)
END IF
pictureNumber = 0
CATCH
    Message.Warning("Error selecting images from: \n\n\t" &
➔ Dialog.Path & "\n\n" & ERROR.Text)
END
```

### [Continued Below](#)

Although we call the `NextPicture` procedure several time it is not where we do the real work of drawing the current image is done. Rather this procedure checks we have valid input for the `DisplayImage` procedure. First we make sure we have some images to display. The `pictureFiles` array holds the paths to the images files in our show. By making sure the `Count` property is greater than zero we ensure there is something to display. We then make sure our index for the current picture is within a valid range. If our index is greater than number of images available then we set it to the start of the show. Similarly if our index is less than zero then we set it to the end of the show. This way we can safely increment and decrement the `pictureNumber` index in the procedures above. We know that here we shall wrap the index around the ends of the array.



Remember that the index in a array starts at zero and the final element of an array is one less then the Count property. When we have a valid index we call the `DisplayImage` procedure that does all the actual drawing work.

FormImageShow.class - Continued

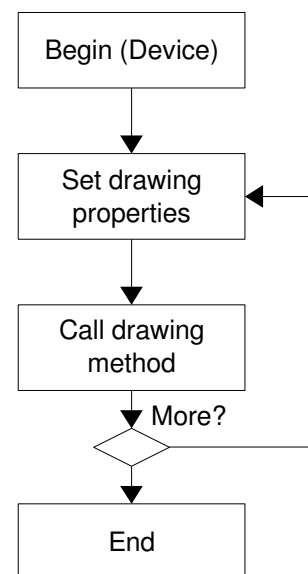
```
PRIVATE SUB NextPicture()  
  ' Check we have some image file paths  
  IF pictureFiles.Count > 0 THEN  
    ' Limit the image number to the number of  
    ' files in our array of file paths  
    IF pictureNumber >= pictureFiles.Count THEN  
      pictureNumber = 0  
    ELSE IF pictureNumber < 0 THEN  
      pictureNumber = pictureFiles.Count - 1  
    END IF  
    ' Now draw the image  
    DisplayImage(pictureFiles[pictureNumber])  
  END IF  
END
```

[Continued Below](#)

Drawing the image on the DrawingArea is very easy. Three objects are involved in this process. The Image object, the Draw object and the DrawingArea control.

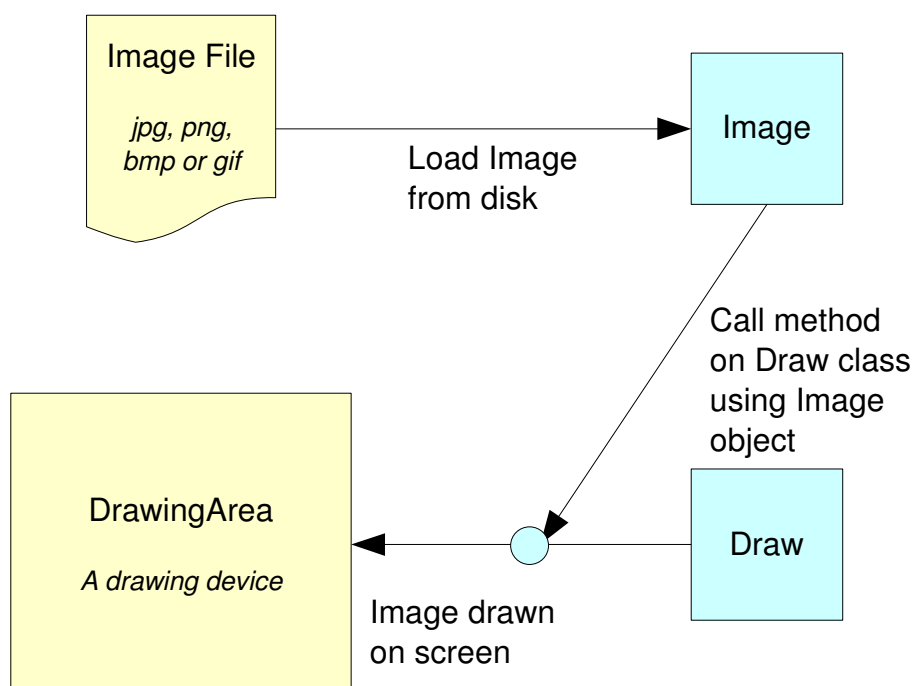
- **Image:** This class stores a bitmap type of image such as a **jpg** or **png** format image. The most important method for us is the stretch method which allows us to resize the image if required.

- **Draw:** You draw on a device. The device can be any of the following objects: Picture, Window (i.e. a Form), the Printer, a Drawing, or a DrawingArea. You start drawing by initializing the device you want to draw on. This is done passing the drawing device to the `Begin` method. You then set the properties required for the next drawing action. This includes properties like selecting colors, fonts or line styles. You then call the method to perform the drawing action. The drawing methods include drawing lines, rectangles, text and images. The process of setting drawing properties and calling drawing methods is continued until the drawing is finished. Finally you call the `End` method to complete the drawing.



- **DrawingArea:** This control acts like visual device that we can draw on. If the `Cached` property is set to `TRUE` then redrawing the current image is handled automatically when the control needs to be refreshed. This is the main reason we use a `DrawingArea`. You can draw directly on a form, but then you need to handle refreshing the current image when required.

This diagram shows the relationship between the drawing objects in our application.



When we first load the image from file. We need to put some error handling on this. The load

method uses the file extension to determine what kind of image to load. If the file content does not match the file extension then we have an error. Even if the user selected some valid images it is possible we may not have read permission on the file. This again could give us an error. If we get an error we simply return from the procedure.

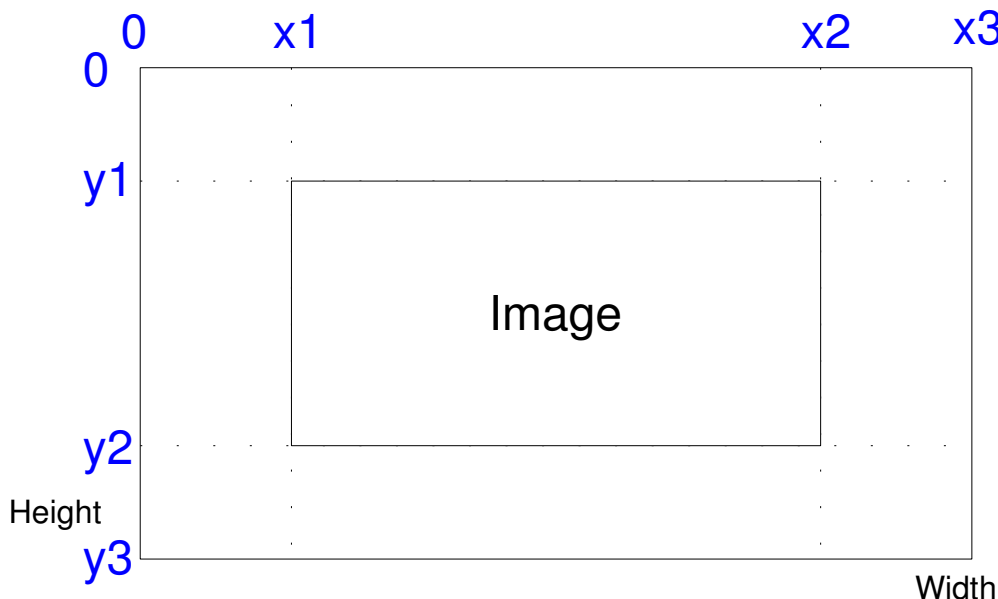
We then check the size of the image. If the image is larger than the size of the screen then we want to scale it to fit on the screen. When scaling we want to scale *isotropically* which means scaling the image by the same factor in both directions. This will keep the aspect ratio of the image. There are two possible scaling factors we could use. Scaling to make the width fit on the screen or scaling to make the height fit on the screen. We take the smallest of these and apply them to both the width and the height of the image.

A more interesting problem is making sure the previous image is cleared. When a new image is drawn it may not be the same size as the previous image. Either in the width direction or the height direction. If all we do is draw the new image we could leave bits of the old image on the screen. This would not look very good.

The first solution I tried was simply to use the Clear method on the DrawingArea and then draw the image. In many cases this works fine. The Clear method clears the DrawingArea using the Background color. The problem was for drawing large images the screen produced a noticeable 'flash' each time an image changed. You saw the background color flash over the entire screen for a fraction of a second. This did not look very good so the Clear method had to go.

The next solution I tried was to draw a solid rectangle over the entire DrawingArea using the background color. Then I would draw the image centred on the screen. This worked better. But you still saw a noticeable 'flash' when a large image was drawn.

The next solution I tried was to draw the new image first. Then to draw over the rest of the screen with the background color to clear any parts of the old image left behind. To do this we need to draw 4 solid rectangles in the background color on each edge of the image. The following diagram will help us calculate the size of the rectangles we need to draw.



Using this this we can see the rectangles we need to draw to clear any possible fragments of the previous image.

<i>Rectangle</i>	<i>Start Point</i>	<i>Width</i>	<i>Height</i>	<i>Draw Rectangle</i>
Top	0, 0	x3	y1	Rect(0, 0, x3, y1)
Bottom	0, y2	x3	y3 - y2	Rect(0, y2, x3, y1) Note that: $y1 = y3 - y2$
Left	0, 0	x1	y3	Rect(0, 0, x1, y3)
Right	x2, 0	x3 - x2	y3	Rect(x2, 0, x1, y3) Note that: $x1 = x3 - x2$

(Our rectangles overlap in the corners. You could make the application more efficient by reducing the size of some of the rectangle. But I do not think it is worth it. I doubt the user would ever notice a speed difference. Compared with the time taken to load, scale and draw the image, drawing the rectangles is very fast. Also you run the risk of leaving lines that have not been cleared across the screen at the joins of rectangles. This could occur if there are any rounding errors in your calculations.)

The final method that might be tried is to use a Picture object as a drawing buffer. With this method you would create a Picture object the size of the DrawingArea. Set the Picture objects background to the DrawingAreas background color. Then draw the image centred on the Picture object. The final step would be to draw the Picture object on to the DrawingArea. Again this works. But it did not provide a solution any better than the previous method we looked at. So for the final version of the

code we have used that method of clearing any previous image. The code listing for drawing and image follows.



Some version of Gambas may require the the line:

```
TRY currentImage.Load(ImagePath)
```

to be:

```
TRY currentImage = Image.Load(ImagePath)
```

as the syntax of the load function has changed.

#### FormImageShow.class - Continued

```
PRIVATE SUB DisplayImage(ImagePath AS String)
    DIM currentImage AS NEW Image
    DIM scale AS Float
    DIM x1 AS Integer
    DIM x2 AS Integer
    DIM x3 AS Integer
    DIM y1 AS Integer
    DIM y2 AS Integer
    DIM y3 AS Integer
    ' Load the image
    ' Some version of Gambas may require this line to read
    'TRY currentImage = Image.Load(ImagePath)
    TRY currentImage.Load(ImagePath)
    IF ERROR THEN RETURN ' If we can not load the image then give up
    ' Check if the image is larger than the screen
    IF (currentImage.Width > DrawingAreaImage.ClientWidth) OR
        (currentImage.Height > DrawingAreaImage.ClientHeight) THEN
        ' Scale image to fit on the screen
        scale = Min(DrawingAreaImage.ClientWidth / currentImage.Width,
        ↳ DrawingAreaImage.ClientHeight / currentImage.Height)
        currentImage = currentImage.Stretch(currentImage.Width *
        ↳ scale, currentImage.Height * scale)
    END IF
    ' Calc rectangles points
    x1 = (DrawingAreaImage.ClientWidth - currentImage.Width) / 2
```

```
x2 = x1 + currentImage.Width
x3 = DrawingAreaImage.ClientWidth
y1 = (DrawingAreaImage.ClientHeight - currentImage.Height) / 2
y2 = y1 + currentImage.Height
y3 = DrawingAreaImage.ClientHeight
' Draw image
Draw.Begin(DrawingAreaImage)
Draw.Image(currentImage, x1, y1)
' Draw rectangles over the area not covered by the new image
Draw.BackColor = DrawingAreaImage.BackColor
Draw.ForeColor = DrawingAreaImage.BackColor
Draw.FillColor = DrawingAreaImage.BackColor
Draw.FillStyle = Fill.Solid
' Draw top rectangle
Draw.Rect(0, 0, x3, y1)
' Draw bottom rectangle
Draw.Rect(0, y2, x3, y1)
' Draw left rectangle
Draw.Rect(0, 0, x1, y3)
' Draw right rectangle
Draw.Rect(x2, 0, x1, y3)
Draw.End
END
```

### [Continued Below](#)

The final procedure we have draws an information message about the slide show. This is called when the user presses the **I** key. First we build the string for the message. We add some information about if the show is running or paused. If it is running we add information about the time interval between images. We then show some information about the current image. We show the current picture number and the total number of images as well as the image path. Note that we add one to the current image index because the string array index for the image paths starts at zero.

Next we calculate the screen position for the message. We want to place the message at the bottom of the screen and in the centre. We will draw this message using the default font. We use the `TextWidth` and `TextHeight` methods on the `Draw` object to get the size of the text. We can then calculate the position for the text. The left position for the text will be half the width of the screen

minus the width of the text message. The top position for the text will be the height of the screen minus the height of the text.

We want to make sure the text is visible. If we simply drew the text it could be over any color that forms the background. So first we draw a filled rectangle as the background. Then we draw the text on top of this rectangle at the position we have calculated.

#### FormImageShow.class - Continued

```
PRIVATE SUB ShowMessage()  
    DIM message AS String  
    DIM x1 AS Integer  
    DIM y1 AS Integer  
    DIM messageWidth AS Integer  
    DIM messageHeight AS Integer  
    ' Get the information message  
    IF TimerShow.Enabled THEN  
        message = "Show Running (Every " & CInt(TimerShow.Delay /  
➡ 1000) & " seconds) "  
    ELSE  
        message = "Show Paused: "  
    END IF  
    IF pictureFiles.Count > 0 THEN  
        message &= "Picture: (" & (pictureNumber + 1) & "/" &  
➡ pictureFiles.Count & ") " & pictureFiles[pictureNumber]  
    ELSE  
        message &= "No pictures"  
    END IF  
    ' Draw the information message  
    Draw.Begin(DrawingAreaImage)  
    ' Calc message position  
    messageWidth = Draw.TextWidth(message)  
    messageHeight = Draw.TextHeight(message)  
    x1 = (DrawingAreaImage.ClientWidth - messageWidth) / 2 ' Center  
➡ of screen  
    y1 = DrawingAreaImage.ClientHeight - messageHeight ' Bottom of
```

```

↳ screen
  ' Draw the message background
  Draw.ForeColor = Color.LightGray
  Draw.FillColor = Color.LightGray
  Draw.FillStyle = Fill.Solid
  Draw.Rect(x1, y1, messageWidth, messageHeight)
  ' Draw the message text
  Draw.ForeColor = Color.Black
  Draw.Text(message, x1, y1)
  Draw.End
END

```

### 3.4: Running the project

The project is now complete so let's run it. You run the project by clicking on the green **Run** button in the project manager window or by pressing the **F5** key. A dialogue should open asking you to select a directory. Select a directory with some images in **png** or **jpg** format. Watch the show!

We now need to test the application works as expected. So run the application and select a image directory. Then press the **D** key to change the image show directory. Test the application with the following kinds of directories:

<i>Open Image Directory Tests</i>		
<i>Test</i>	<i>Expected Result</i>	<i>Pass/Fail</i>
Open a directory with images in it.	Image show displayed.	
Try an empty directory.	A no images found message.	
Try a directory with no images but some other file types.	A no images found message.	
A directory with some jpg image files.	Image show displayed.	
A directory with some jpeg image files.	Image show displayed.	
A directory with some png image files.	Image show displayed.	
A directory with some gif image files.	Image show displayed.	
A directory with some bmp image files.	Image show displayed.	



**Open Image Directory Tests**

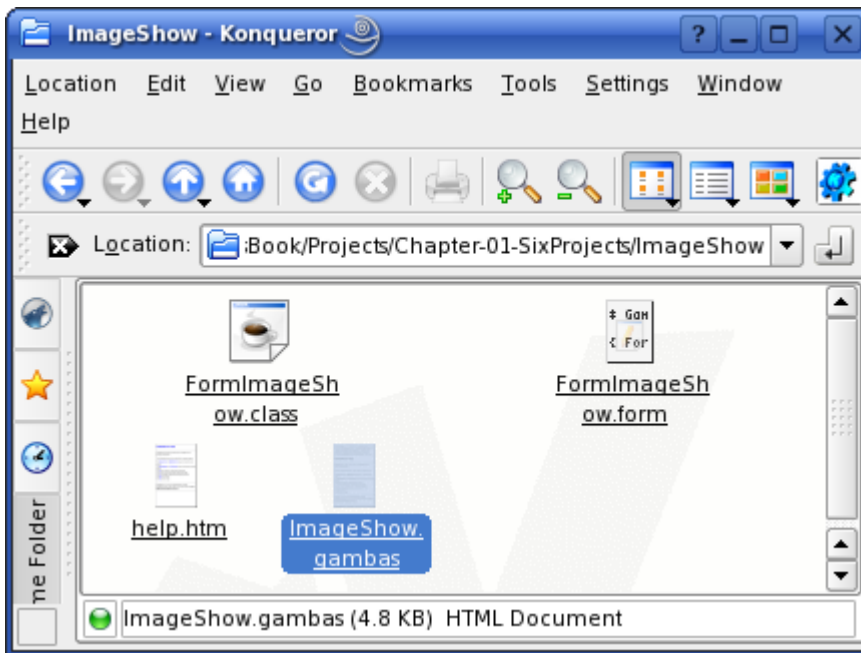
<i>Test</i>	<i>Expected Result</i>	<i>Pass/Fail</i>
A directory with some xpm image files.	Image show displayed.	

Make sure in all of the above tests that the application does not try to open any files that are not images and that all the images are shown. The application should not crash and you should be able to select another directory. We now will test all the other keys perform the correct action.

**Key Press Tests**

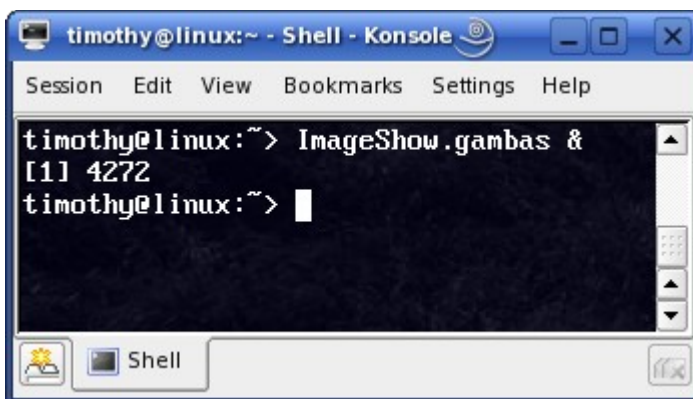
<i>Test</i>	<i>Expected Result</i>	<i>Pass/Fail</i>
With an image show press the <i>space bar</i> .	Next image should be displayed.	
With an image show press the <i>right arrow</i> .	Next image should be displayed.	
With an image show press the <i>backspace</i> key.	Previous image should be displayed.	
With an image show press the <i>left arrow</i> .	Previous image should be displayed.	
With the show running press the <i>S</i> key.	The show should pause. (Wait more than 15 seconds to make sure.)	
With the show paused press the <i>S</i> key.	The next image should be displayed and the show start running.	
With the show running press the <i>I</i> key.	An information line should be shown. It should say the show is running and give the current image name.	
With the show paused press the <i>I</i> key.	An information line should be shown. It should say the show is paused and give the current image name.	
With the show running press the <i>F1</i> key.	A help message should be displayed and also the mouse. When the message is cleared the mouse should be hidden.	
Press the <i>Esc</i> key.	The application should close.	

Finally for this project we are going to create an executable version of the application. This way you can run ImageShow outside the Gambas development environment. First make sure you have completed all the testing described above. Then in the Gambas project manager select the **Project** menu and then the **Make executable...** sub menu. This will show a standard dialogue asking for the name of a file. Accept the defaults and click the **OK** button.



This will create an executable version of your application in your project directory called `ImageShow.gambas`.

In Unix/Linux your home `bin` directory is a good place to store your personal applications and scripts. On most systems this directory is in your `PATH` so it will be searched when you enter commands from a terminal. Move the file **ImageShow.gambas** that we have just created in your project directory to your **\$HOME/bin** directory.



Now open a terminal window and type `ImageShow.gambas &` and our application should start. The `&` after the command is so the terminal does not wait for the `ImageShow` process to end. It is not required to start `ImageShow`.



Not all Linux distributions have a **\$HOME/bin** directory.

On some distributions you can `cd` to the directory where the executable contained. Then type `./ImageShow.gambas` and the executable should run.

On some distributions you need to give the full path to the executable. If the executable was in a directory called `Images` in our home directory then try:  
**\$HOME/Images/ImageShow.gambas**

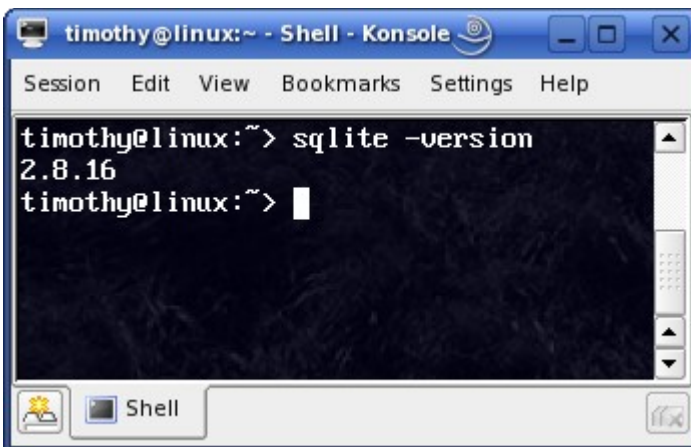
One reason I wanted to create a executable for this application was to demonstrate that files in the data section of a Gambas project are included in the final executable. And that they use the relative path within the application for their file location. To prove this works press the **F1** key and view the help. This should correctly load the HTML help file we created and load it into the message box for display.

You could also add a link to this executable on your desktop. This would allow you to run our application by clicking on an icon. How you do this is dependent on the window manager you are using. See the help for your window manager.

---

## 4: Database Project: Notations

Microsoft Visual Basic became very popular on the Windows platform. One reason for this success is that Microsoft Visual Basic was a good front end for database applications. Many users, like myself, coming from the Windows world to Linux will have experience of developing applications with Visual Basic to act as a user interface to a database. They will want to transfer these skills to Linux. In this example we are going to get you started with a small project to communicate with an SQLite 2 database. With Gambas you can use almost the same code to communicate with a MySQL or PostgreSQL database. With later versions of Gambas you can also connect to a SQLite 3, ODBC or Firebird database.



```
timothy@linux:~> sqlite -version
2.8.16
timothy@linux:~> 
```

As with installing Gambas this tutorial will not cover installing SQLite. SQLite should be installed with most Linux distributions. In order to check you have SQLite installed on your workstation open a terminal and type `sqlite -version`. If a version number is returned then SQLite is installed.

Note that this will check if version 2 of SQLite is installed. We are using version 2 because it is supported by more versions of Gambas. Newer version of Gambas also work with SQLite version 3. You can check for SQLite version 3 with `sqlite3 -version`.

If SQLite is not installed then you should be able to install it through your package manager or visit the SQLite web site at <http://www.sqlite.org/>. If you are new to SQLite it is also worth looking at the documentation on this web site.

There are two possible approaches you could take to interfacing with a database in Gambas. The first is to use the Gambas database objects. With this method your code would perform the following actions:

- Create the database in Gambas using the connection object.
- Build the database tables, indexes, etc. using the Gambas database objects.
- Read data from the database using the Result object. You would create the Result object using the Find or Edit method on the database connection.
- Update and Delete actions on records would be through the Result object. You would create

the Result object using the Edit method on the connection.

- Adding records would be through the Result object. You would create the Result object using the Create method on a connection.

This approach has the following advantages:

- You are abstracted from the database layer.
- You do not need to know much SQL (Structured Query Language).
- Your code is portable between the databases supported by Gambas. You should only need to change the connection parameters to switch database types.

It also has a few disadvantages:

- Search and filtering data is more awkward and is less well supported than directly using SQL.
- Views and database queries requiring more than one table are not as well supported as in directly using SQL.
- There are some features of the underlying database that you can not access.

For most kinds of database application the above method is the best. However there is a second method you could use. With this method you would be to use the Exec method on the connection object to send SQL statements to the database. With this method your code would perform the following actions:

- There are two possible method you could use to create the database.
  - Create the database using tools provided by the database vendor or a third party.
  - Create the database in Gambas and build the tables, indexes etc. using a SQL script. Send this script to the connection Exec method.
- You would read data from the database using the Exec connection method using a SQL query and return this data to a read only Result object.
- Update, Insert and Delete should also be preformed using the SQL statements sent to the Exec connection method.

The advantages are:

- You gain fine grained control over the database. You control the detail of each SQL query or command.
- Most databases support more field types for tables than are supported by Gambas.
- Features in the database not supported in Gambas are easily accessible.
- It is easy to access database views and multi table queries.

Possible disadvantages are:

- You need a good knowledge of SQL for your target database.

- You need to format the data yourself when sending or receiving it from the database. For example you need to make sure dates are in a format understood by the database.
- You need more careful error handling to catch SQL errors. Debugging the SQL is also more complex.
- Your application is properly not portable if you need to switch database types.

Which method should you chose? My general advice would be:

- If you only require a small database or you need portability between databases then stick to the first method. In this case try to avoid using the Exec method of obtaining data or passing SQL queries to the database.
- If you have a large database and the database type is not likely to change then consider the second method. Also consider the second method if your database queries are going to be large or complex. If you select the second method only use the Connection and Result objects and only use the Exec method to perform queries.

I would not mix the two approaches. Of course there is nothing to stop you if you really want to. But you could end up with none of the advantages of either method and all of the disadvantages.

In this tutorial we are going to demonstrate the first of these methods. This is after all a tutorial about Gambas and not SQL. Also we can then demonstrate more of the Gambas database objects you might want to use.



In a [Appendix](#) to this tutorial we shall look at how to convert this Notations example to the second approach using Exec and SQL statements.

A common form of interface design for a database is the Master/Detail design. Here you have a list of records in a window. There is some field from the database that summarises the database records and this is used to create a list. When a record from the list is selected its details are displayed in the rest of the window. If required the details can be updated and saved to the database. There will also be some mechanism for adding and deleting records. This Notations example will follow this basic interface design to a database application.

With this example we are going to create a note taking application. We shall store the notes in a SQLite database. The user should be able to:

- Add notes.
- View notes.
- Update notes.
- Delete Notes.
- Cut, Copy and Paste text from the Clipboard to and from notes.

- View a list of note titles. You can then click on a note title to view the complete note.
- Search the text in note titles and notes for a given search text.

We shall keep our database very simple. The database name is going to be the same as the application name. We shall only need one table and this will be called “Notes”. The following entry lists the fields in the table.

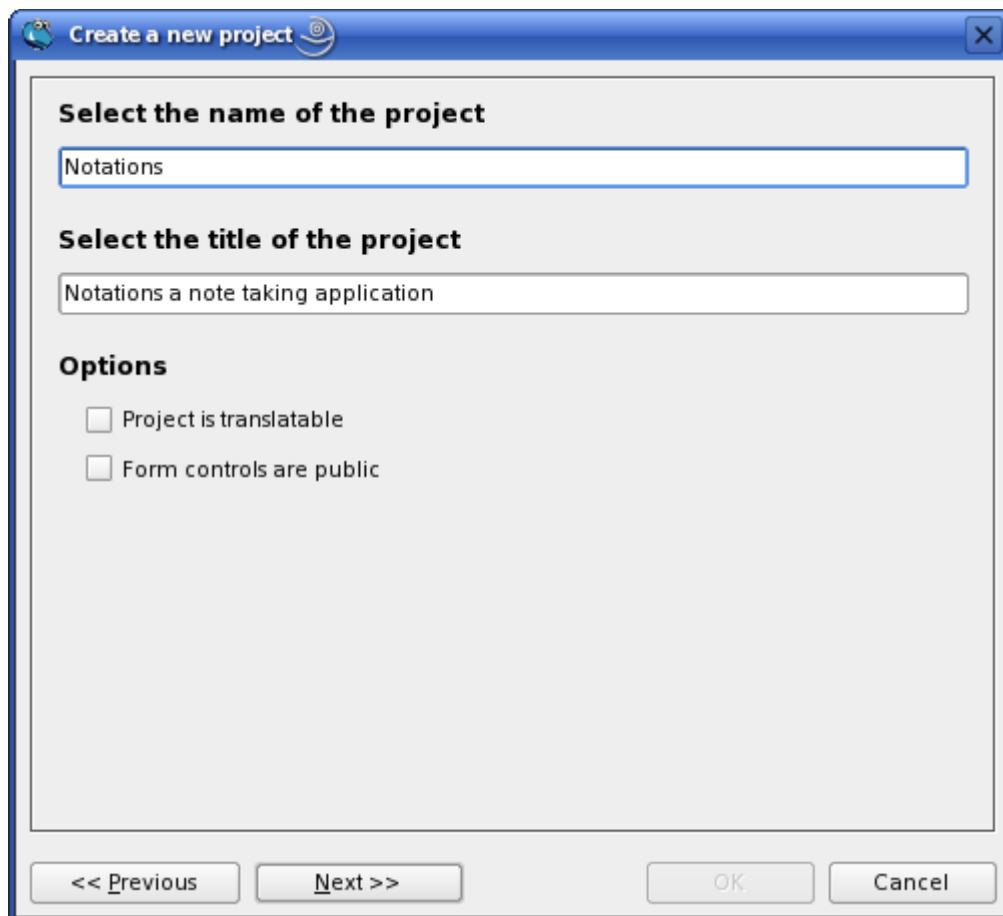
<i>Field Name</i>	<i>Type</i>	<i>Comment</i>
CreateDate	DATETIME	Primary Key. When a new note is first created this will be set to the current date and time.
LastModified	DATETIME	When a note is first created this will have the same value as the CreateDate field. Each time a note is updated this field will be set to the current time.
Title	STRING	Length 0 for a unlimited string
Note	STRING	Length 0 for a unlimited string
Priority	INTEGER	Priority encoded as an integer. 0 is low, 1 is medium and 3 is high priority.



*Notations* is named after a piece of music by the French composer Pierre Boulez originally written in 1945. This early work was a set of 12 short pieces for piano. Later, in 1978, Boulez orchestrated four of the pieces for large orchestra. He revised them again in 1984. Then added an additional orchestration in 1997. Given the name, the history of the music and the fact that Boulez is French and so is the main developer of Gambas, Benoit Minisini, this seemed a good name for a note taking application.

## 4.1: Creating the project

The first step is to create a new Gambas project. So open Gambas and select **New project....** This will start the new project wizard. The first page of the wizard simply shows a welcome screen with details of how to use the new project wizard. Click the **Next >>** button to for the next page of the wizard. This shows a page where we can select the type of Gambas project we want to build. We are going to use the default option of a graphical project. So click the **Next >>** button. This shows a page when you enter the name and title for your project we are going to create.

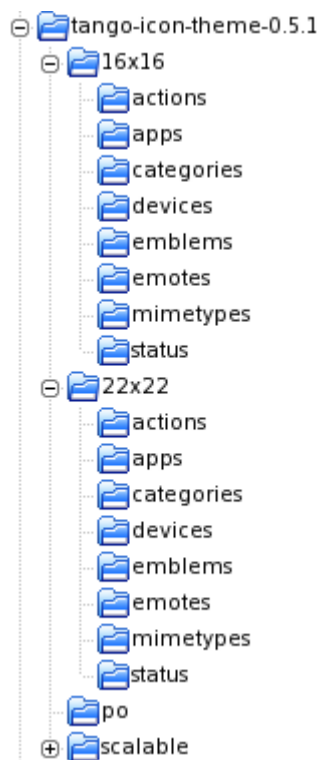


Give the project the name `Notations` and the title `Notations a note taking application`. Leave the other options blank. Click on the **Next >>** button. Select the location where you want to save the project. Then click the **Next >>** button. This final page lists the options you have selected for the project. Check through the options and then click the **OK** button. We have created a new Visual Basic project and Gambas will open showing the Gambas project manager.

With this project we are going to use a tool bar to display some buttons. Tool bars can provide a nice looking interface to the user and take up little space. So we shall need some icons for these buttons. The Tango project aims to provide a common look and feel to the desktop. It also provide a good icon set for use in your applications. You can get the required icons from the example applications that come with this tutorial or the Tango web site <http://tango-project.org/>. This site also provides useful information about designing and using icons.

Go to the project directory and create a sub directory called `Images`. In this directory place the following icons from the Tango icon set.

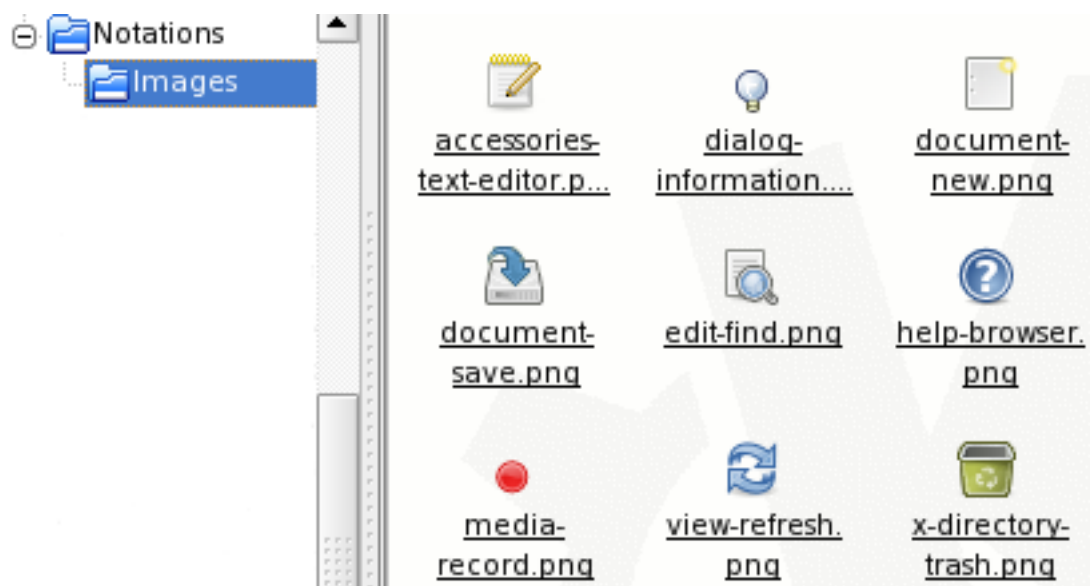




The tango icons are grouped into size and then category, Select the icons shown in the following table and place them in the Images directory.

<i>Icon Size</i>	<i>Icon Category</i>	<i>Icon Name</i>
16 x 16	actions	dialog.information.png
16 x 16	status	media-record.png
22 x 22	actions	document-new.png document-save.png edit-find.png view-refresh.png
22 x 22	apps	accessories-text-editor.png help-browser.png
22 x 22	mimetypes	x-directory-trash.png

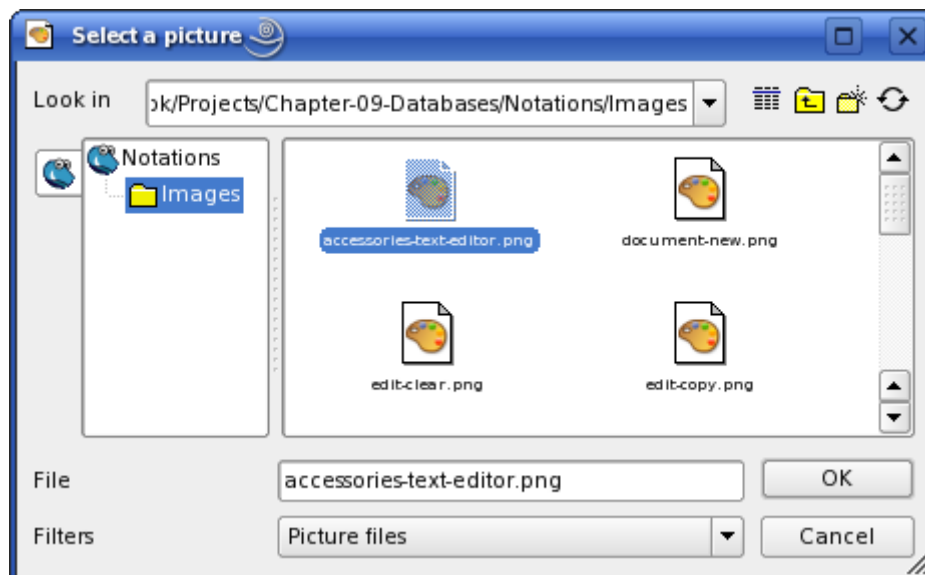
The following screen shot shows how the Images directory should look after copying these icons.



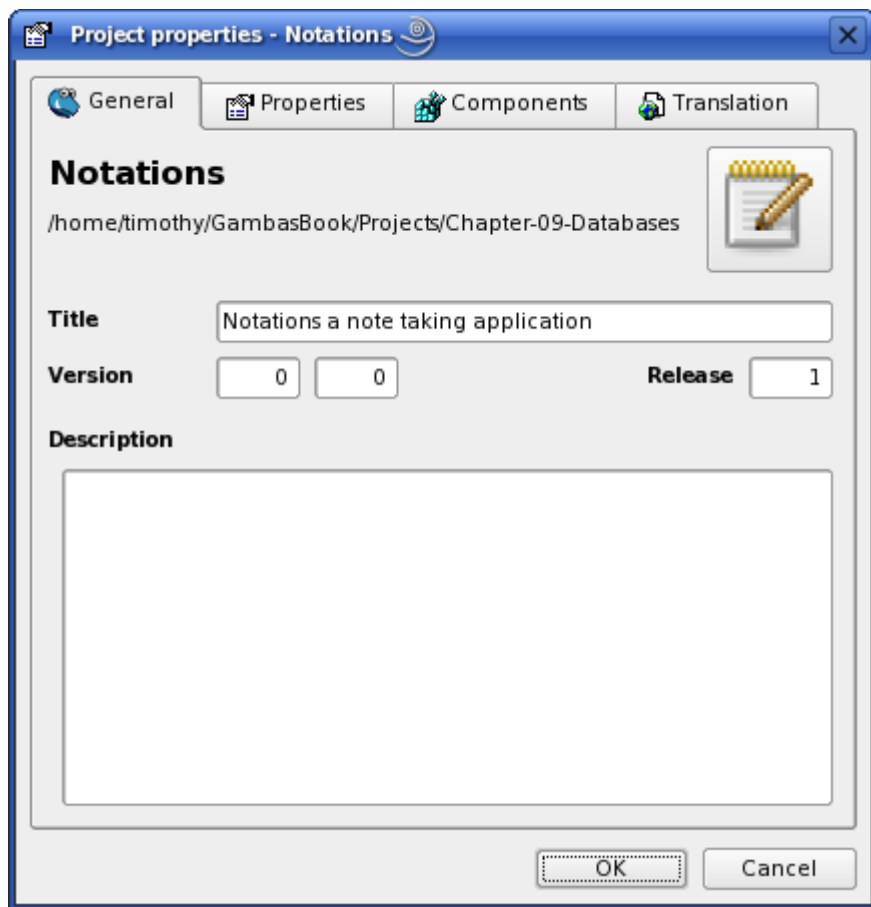
From the project manager window open the **Project** menu and then the **Properties...** sub menu.



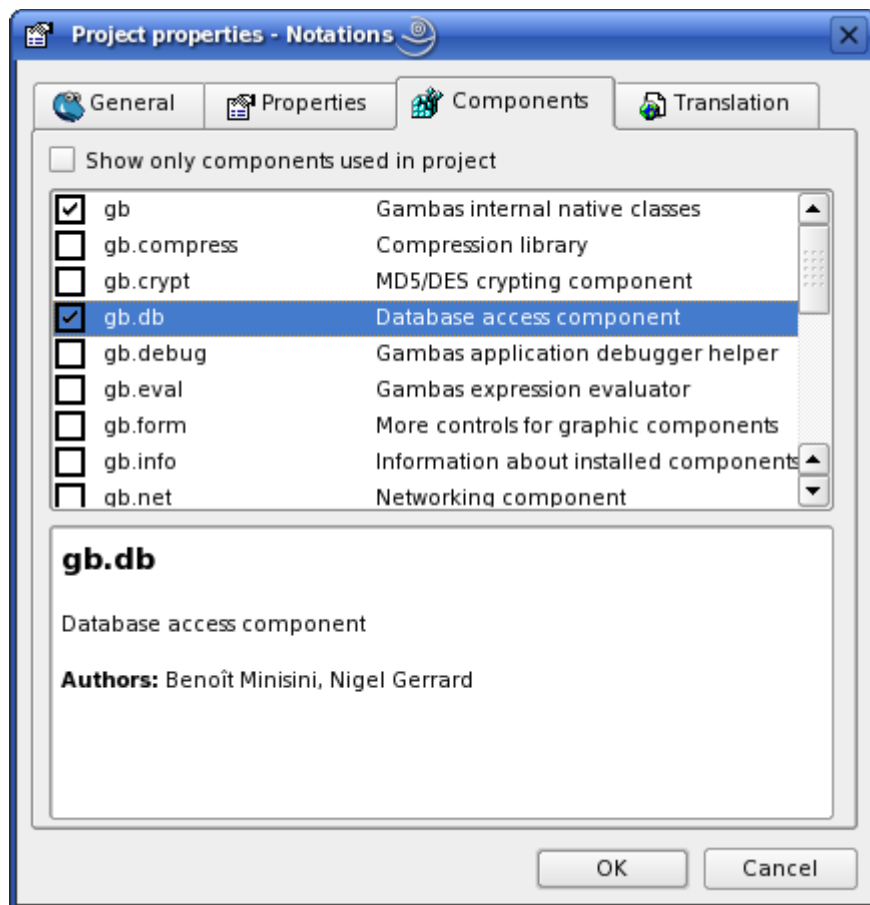
This shows the project properties dialog. Select the Gambas icon and then select the Images/accessories-text-editor.png icon from the Images directory we created above.



Click on the OK button. The Project Properties should now show the accessories-text-editor.png icon.



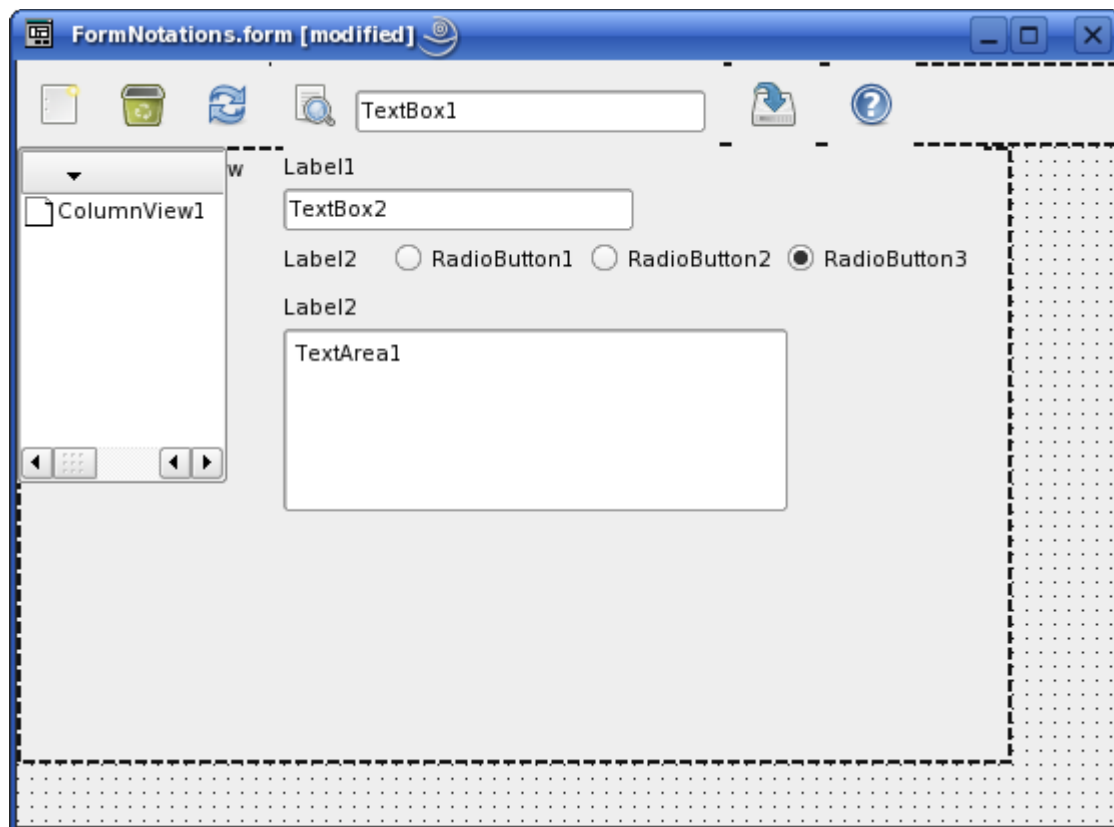
In the previous two applications we only used the standard Gambas components. However there are many more controls and components available. We are going to add the **gb.db - Database access component** to the project. These will give us access to the Gambas database objects and provide the interface to the database. Select the Components tab on the previous dialog.



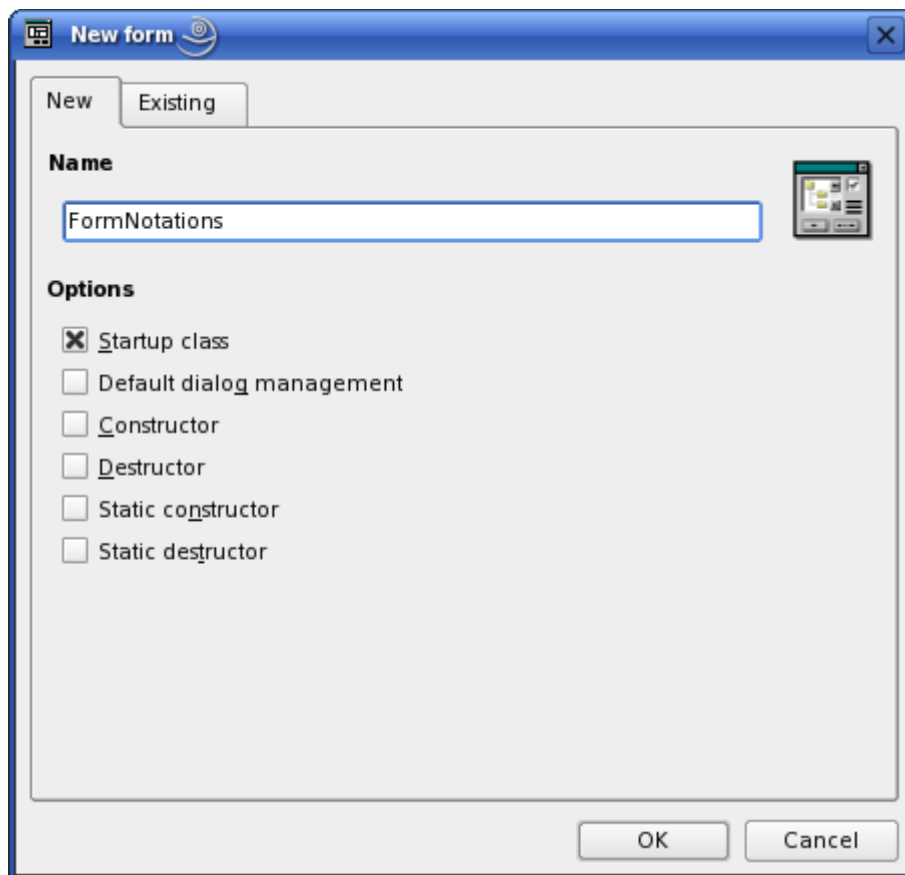
Make sure the **gb.db - Database access component** and item is checked. Then click the **OK** button to accept this dialog.

## 4.2: Creating the user interface

This application is only going to have one window as the interface. This interface is going to be slightly more complex than the previous two examples. In order to give us some idea of what we are working towards here is a screen shot of the look of the form when we have added all the controls and it is in design mode.



Click in the project manager window and create a new form. Call the form `FormNotations` and accept the option for this to be a startup class.



We are first going to create some buttons in a tool bar to go across the top of the form. So select the **Container** tab in the Gambas toolbox. From this select the **HPanel** and add this to the top of the form. Resize this control so it is 42 pixels high and the width is a bit less than the width of the form. The exact width does not matter as this will be controlled at runtime.

To resize controls first click on the required control. This will show eight resizing boxes. Then click and drag one of these resizing boxes with the mouse. If you have the Gambas properties window open you will see the Height and Width properties change. You can also edit these properties directly in this window.



In this project we are going to add quite a few controls that need to be placed inside a parent control. It is all too easy to place the new control *above* the parent container rather than *inside* the parent container. The form will still look correct in design mode but unfortunately the controls will not resize correctly at run time. Try to follow this little sequence each time you add a child control to a parent container control:

- Click on the parent control. Make sure it is highlighted and its eight resizing boxes are visible.
- Then click on the tool box control you are going to add.
- When you click and drag the mouse to add the control make sure you are inside the required parent control. Also make sure the eight resizing boxes are still

visible.

It is a good idea every so often to move the parent control you are adding child controls to. Then move it back to its desired position on the form. When you do this check all child controls move with the parent. If any child control does not move with its parent then it is properly not placed correctly inside the parent control. Try adding the the misplaced child control again.

The next step is to add the buttons we want in our tool bar. Select the **Form** tab in the Gambas toolbox. The first button is going to be our new note button. Now click on the **ToolButton** and add this to the **HPanel** we have just added. Make sure this button is *inside* the HPanel and not *above* the HPanel. Move this button so it is at the far left of the HPanel and is 42 pixels high and 42 pixels wide. Now add two more **ToolButton** to the **HPanel** these are going to be our Delete button and our Refresh button. Move each button so it is 42 pixels high and 42 pixels wide and placed next to the previously added button. We should now have three buttons in the HPanel.

The next button to add is the Search button. We are also going to add a **TextBox** next to this button in which to place the notes search text. When the HPanel is resized at run time it will move controls onto an additional line if they do not fit on the width of the control. This is nice behaviour and means we can keep all of our buttons visible no matter how wide the user resizes our form window. What would also be nice is if the Search button and **TextBox** were always next to each other when this resizing takes place at run time. We can achieve this by placing these two controls inside a **Panel** which is inside our HPanel. This search panel will not be visible at run time.

So lets add these Search controls. Select the **Containers** tab in the Gambas toolbox. From this select the **Panel** and add this to the **HPanel**. Move this **Panel** next to our Refresh button and set its height to 42 pixels and its width to about 224 pixels. Select the **Form** tab in the Gambas toolbox. Add a **ToolButton** inside this search **Panel**. Move it to the left of the search **Panel** and resize the button so it is 42 pixels high and 42 pixels wide. Now add a **TextBox** to the search **Panel**. Resize this text box so its height is 21 and its width is 168. Also make sure its X value is 42 and its Y value is 14. You may need to use the Gambas properties window to change some of these values.

The final part of our tool bar is two more buttons. These will be for updating a note and a help dialog button. Select the **Form** tab in the Gambas toolbox. Place two **ToolButtons** inside the **HPanel** next to the search **Panel**. Resize each button so it is 42 pixels high and 42 pixels wide.

That has added all the controls we need for our tool bar. Now we shall add the controls that form the main part of our window and display our notes. On the left side of the form window we are going to have a list of notes. On the right side we are going to have the details of the currently selected note in the list. In order for the user to resize each half of our form window we are going to use a **HSplit** control. Select the **Containers** tab in the Gambas toolbox. Add a **HSplit** to the form under the HPanel. We shall be adding more controls to the **HSplit** so make it large enough for this. Its exact size does not matter as it will be resized at run time.



In Gambas version 1 the HSplit control is not available so we shall have to miss it out. This means the user will not be able to resize the two half's of the window with the central bar. Place the ColumnView we add next directly on the form. Also we directly place second panel we add later directly on to the form.

We are going to use a ColumnView to provide the list of notes. At run time we shall set the column titles and content for this list. So select the **Form** tab in the Gambas toolbox. Then add a **ColumnView** to the **HSplit** and make sure it is inside the HSplit. Push the ColumnView to the top left corner of the HSplit and make its width about a quarter of the HSplit. One of the nice things about using the HSplit control is that the resizing of the ColumnView will be handled by the HSplit control when it is resized. We shall not have to add and code for this.

This next group of controls is where we shall display a particular note. We first need to add another Panel so the HSplit groups these controls together at run time. Select the **Containers** tab in the Gambas toolbox. Then add a **Panel** to the **HSplit**. Push the panel to the top of the HSplit and next to the ColumnView. Make the size of the Panel covers most of the area of the HSplit not occupied by the ColumnView.

The final set of controls to add will show the details for a particular note. All these controls go inside the Panel we have just added. Select the **Form** tab in the Gambas toolbox. Now add these controls to our Panel.

These next controls are for the title of the note. Add a **Label** to the top of the **Panel**. Under the Label place a **TextBox**.

Now add some controls where the user can select the priority of a note. Add another **Label** under the **TextBox** and then along side this add three **RadioButtons**.

These next controls are for the content of the note. Add another **Label** under the Priority Label and then a **TextArea** under this Label.

We are not done with these controls yet. We need to change some of their properties. So make sure the Gambas properties window is displayed. Click on each control in turn and change the required properties. This is a list of all the properties we need to change. (Gambas 1 users should skip the HSplit1 control.)



<i>Default Name</i>	<i>Property</i>	<i>New Value</i>
FormNotation	Icon	Images/accessories-text-editor.png
	Border	Resizable
HPanel1	Name	HPanelTools
ToolButton1	Name	ToolButtonNew
	ToolTip	Create a new note
	Picture	Images/document-new.png
ToolButton2	Name	ToolButtonDelete
	ToolTip	Delete the selected note
	Picture	Images/x-directory-trash.png
ToolButton3	Name	ToolButtonRefresh
	ToolTip	Refresh the notes list
	Picture	Images/view-refresh.png
Panel1	Name	PanelSearch
ToolButton4	Name	ToolButtonSearch
	ToolTip	Search notes for text
	Picture	Images/edit-find.png
TextBox1	Name	TextBoxSearch
	Text	<blank>
ToolButton5	Name	ToolButtonUpdate
	ToolTip	Update current note
	Picture	Images/document-save.png
ToolButton6	Name	ToolButtonHelp
	ToolTip	Display help
	Picture	Images/edit-undo.png
HSplit1	Name	HSplitWindow
ColumnView1	Name	ColumnViewNotes
	Sorted	True
Panel2	Name	PanelNote
Label1	Name	LabelNoteTitle
	Text	Title:
TextBox2	Name	TextBoxTitle
	Text	<blank>

<i>Default Name</i>	<i>Property</i>	<i>New Value</i>
Label2	Name	LabelNotePriority
	Text	Priority:
RadioButton1	Name	RadioButtonHigh
	Group	RadioButtonPriority
	Text	High
RadioButton2	Name	RadioButtonMedium
	Group	RadioButtonPriority
	Text	Medium
RadioButton3	Name	RadioButtonLow
	Group	RadioButtonPriority
	Text	Low
	Value	True
Label3	Name	LabelNoteText
	Text	Note:
TextArea1	Name	TextAreaNote
	Text	<blank>

In order to test our work so far lets add a small amount of resizing code code to the project. Right click on the form and add a `Form_Resize` event. Now right click on the `HSplitWindow` object and add a `HSplitWindow_Resize` event. In these two event handlers add the following code.



Gambas version 1 users would only add the `Form_Resize` event. Also the resizing code is a little different – [see below](#).

```
FormNotations.class
```

```
PUBLIC SUB Form_Resize()
    HPanelTools.Width = ME.ClientWidth
    HSplitWindow.Move(0, HPanelTools.Height, ME.ClientWidth,
    ➔ ME.ClientHeight - HPanelTools.Height)
END

PUBLIC SUB HSplitWindow_Resize()
    TextBoxTitle.Width = PanelNote.ClientWidth
```

```
    TextAreaNote.Resize (PanelNote.ClientWidth,  
↳ PanelNote.ClientHeight - TextAreaNote.Top)  
END
```



Our resizing code is slightly different for Gambas version 1. This is because we had to omit the HSplit control.

```
FormNotations.class  
  
PUBLIC SUB Form_Resize()  
    HPanelTools.Width = ME.ClientWidth  
    ColumnViewNotes.Top = HPanelTools.Height  
    ColumnViewNotes.Height = ME.ClientHeight -  
↳ ColumnViewNotes.Top  
    PanelNote.Top = ColumnViewNotes.Top  
    PanelNote.Width = ME.ClientWidth - PanelNote.Left  
    PanelNote.Height = ColumnViewNotes.Height  
    TextBoxTitle.Width = PanelNote.ClientWidth  
    TextAreaNote.Resize (PanelNote.ClientWidth,  
↳ PanelNote.ClientHeight - TextAreaNote.Top)  
END
```

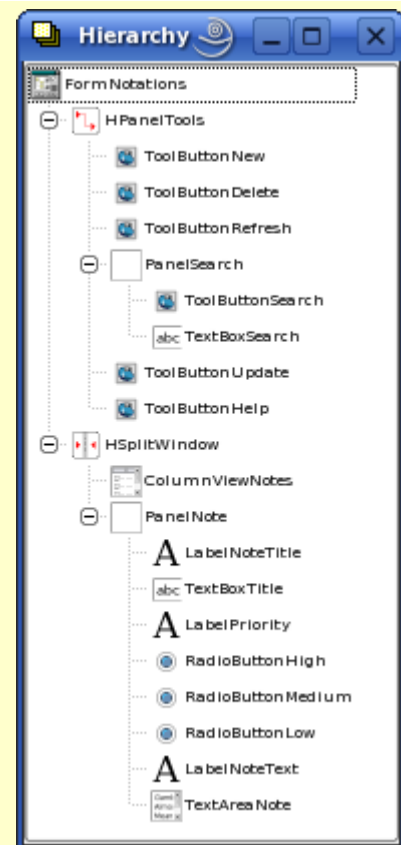
Now it is time to test the work we have done so far. Hit the green run button in the project manager or press the **F5** key. The application will do very little. But you should be able to grab the central bar between the ColumnView and the TextArea panel and the controls will resize themselves correctly. (At least for Gambas 2 users.)



You should also be able to resize the form window and the controls will resize themselves. If they do not then the most likely reason is that some controls are not correctly inside their containers.

You can check this using the Gambas Hierarchy tool. In design mode select the form. Now from the Gambas project manager select the **View** menu and then the **Hierarchy** option or press **Ctrl+H**. Check the controls hierarchy on your form matches the screen shot on the right.

If a control is not at the right level in the hierarchy then click on the offending control. Press **Ctrl+X** to cut the control. Then select the correct parent control and press **Ctrl+V** to paste the control into its correct container.



	Bring to foreground	Home
	Send to background	End
	Horizontal	Ctrl+Right
	Left to right	Ctrl+Shift+Right
	Vertical	Ctrl+Down
	Top to bottom	Ctrl+Shift+Down
	Hierarchy	Ctrl+H

If a control is at the right level in the hierarchy but not in the right order then Right click on the parent control and this will show a popup menu. Select the **Arrangement** option and this will show a sub menu. In this menu select **Left to right**. This will reorder the child control in the order they are on the form.

We want to show some help informations to the user when they click on the help button. We are going to use the same method as the previous project. This method is suitable when the required amount of help is small and will fit on one page. The Gambas message box is quite a flexible class and will take a HTML formatted message. So with the single line of code:

```
Message.Info(File.Load("help.htm"))
```

we can display a HTML file.

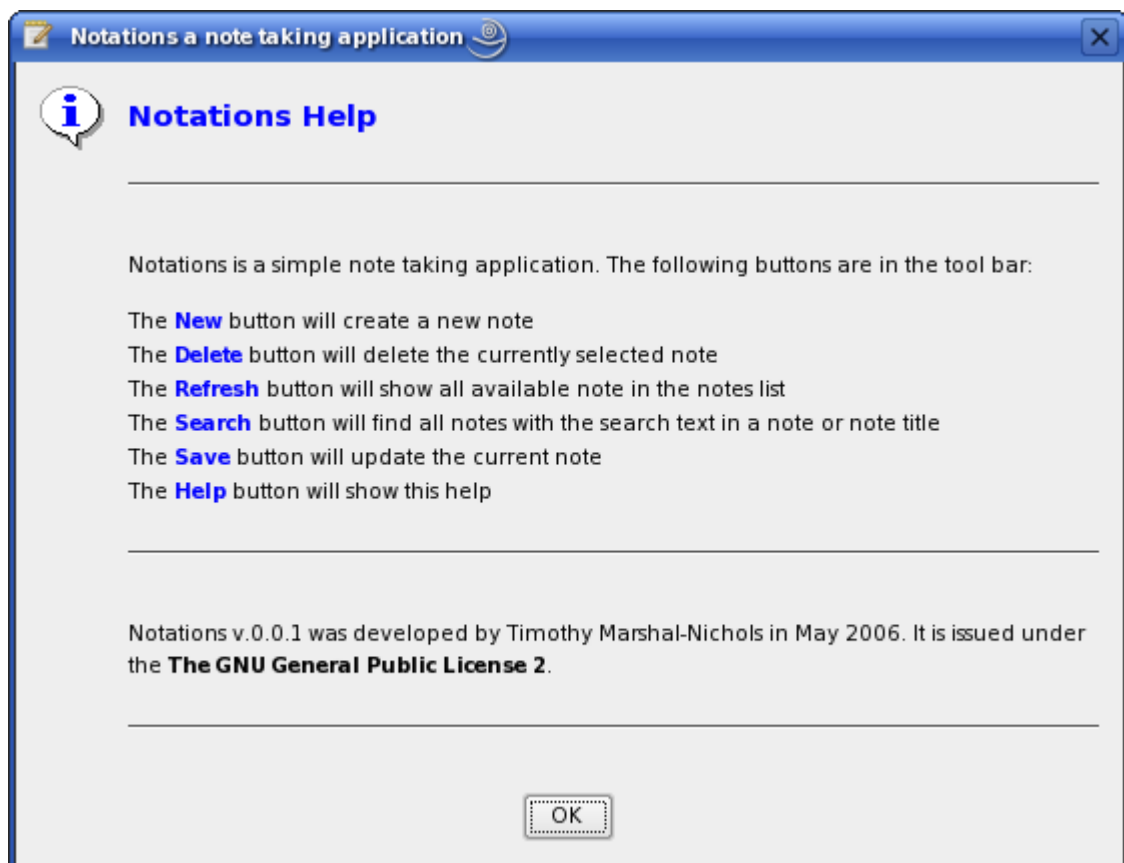
So let create the HTML file in Gambas. Right click in the Gambas project manager. This brings up a pop up menu. From the list select **New**. This shows a sub menu with the items we can add to our project. We need to add a Text file so select **Text file...** from this menu. Give the file the name `help.htm` and press the **OK** button to save it. This will add the file to the data section of your project and will open the new text file in the Gambas text editor window. Enter the following HTML

into the text editor and save it. Note that, as before, the ➞ icon in the following listing means this line is a continuation of the previous line.

help.htm

```
<html>
  <head>
    <title>Notations Help</title>
  </head>
  <body>
    <h2><font color='Blue'>Notations Help</font></h2>
    <hr>
    <p>Notations is a simple note taking application. The
➞ following buttons are in the tool bar:
    </p>
    <p>The <b><font color='Blue'>New</font></b> button will create
➞ a new note
    <br>The <b><font color='Blue'>Delete</font></b> button will
➞ delete the currently selected note
    <br>The <b><font color='Blue'>Refresh</font></b> button will
➞ show all available note in the notes list
    <br>The <b><font color='Blue'>Search</font></b> button will
➞ find all notes with the search text in a note or note title
    <br>The <b><font color='Blue'>Save</font></b> button will
➞ update the current note
    <br>The <b><font color='Blue'>Help</font></b> button will show
➞ this help
    </p>
    <hr>
    <p>Notations v.0.0.1 was developed by Timothy Marshal-Nichols
➞ in May 2006. It is issued under the <b>The GNU General Public
➞ License 2</b>.
    </p>
    <hr>
  </body>
</html>
```

The following screen shot shows what this HTML looks like in a message box when the application is run and the user presses the help button.



There is one final file we need. When we create the notes database we shall also add a first note. This will be a welcome note giving the user some help information. As the text for this note is going to be more than a few line we are going to load the text from a file. This method of loading the text from a file will be very similar to that used for the help. This also gives us a great deal of flexibility if we need to change this application in the future. We can easily incorporate text changes to the welcome help text.

Right click in the Gambas project manager and select the menu to add a new text file. This time give the file the name `Welcome.txt` and press the **OK** button to save it. This will add the file to the data section of your project and will open the new text file in the Gambas text editor window. Enter the following text and save the file. As you will notice it is very similar to the help text.

`Welcome.txt`

Welcome to Notations. A simple note taking application

The following buttons are in the tool bar:

The New button will create a new note

The Delete will delete the currently selected note

The Refresh will show all available note in the notes list

The Search will find all notes with the search text in a note or  
→ note title

The Update will save the current note

The Help button will show help

Notations v.0.0.1 was developed by Timothy Marshal-Nichols in  
→ May 2006.

It is issued under the The GNU General Public License 2.

So we created all the item we need for the user interface to this application. We have only needed 7 line of code (and 4 of them beginning and end's to procedures) to get a decent interface. It shows that with a little knowledge of how containers and control operate within Gambas you can create some good user interfaces.

### **4.3: *Checking our user interface for CRUD***

Many application developers would already have leapt into the coding stage by now. But it is worth standing back and checking the design of our application for functionality. One checking method often used with database applications is CRUD. CRUD stands for CREATE, READ, UPDATE, DELETE. It is a check list to apply to records or entities in database tables against operations performed by users.

The CRUD check list is often set out as a matrix. Along one side are listed the tables or entities in the database. Along the other side are listed the user roles for the database. Inside each cell you list the CRUD actions that each user role can perform on each table or entity. The following diagram demonstrates an imaginary CRUD matrix.

<i>User</i>	<i>Table 1</i>	<i>Table 2</i>	<i>Table 3</i>
<i>User 1</i>	R,U	R,U	-
<i>User 2</i>	C,R,U,D	C,R,U	R,U
<i>Administrator</i>	C,R,U,D	C,R,U	C,R,U,D

From this matrix we can see that User 1 can only read and update tables 1 and 2. User 2 has different levels of access to each table. The Administrator has full access to all tables apart from table 2. Also for table 2 we have no way of deleting a record. These observations might be correct for our imaginary application. But with a CRUD matrix like this we need to check the requirements for our application again to find out.

CRUD is a check list. It is a check list that helps you spot any missing operations in your application. This does not mean that every user should be able to perform all these operations. In many cases there are operations you would not want performed by all users. For example in a banking database you would not want all users to be able to delete details of financial transactions. Rather with CRUD you are checking the functionality of your database application. You then have to think about why some user should have or *not* have some particular functionality. It is the thinking process that is important.

Often an additional operation is added to the CRUD check list. This is L for LIST. Can a user list database records? I would also add S for SEARCH. Can a user search database records using some search criteria?



There are lots of descriptions of CRUD on the internet. Search for “CRUD+databases”. You could try: [http://en.wikipedia.org/wiki/CRUD\\_\(acronym\)](http://en.wikipedia.org/wiki/CRUD_(acronym))

I find CRUD is a useful check list even when you are not dealing with a database. Any place in a application where you are adding or removing data from controls at run time is worth putting through the CRUD check list. This could be a grid, a text area, a combo box or many other controls.

CRUD is most often presented as a matrix. In our Notations application we only have one user and one table. So the matrix looks a bit silly. Instead lets use the CRUD-LS check list to think about the actions performed by our user on the database records.

- **Create:** The user can create notes using the add button. This will add the current content of the Note Title text box, Priority selection and Note text area to the database.
- **Read:** Clicking on a note in the column view will place that record in the Note text box, Priority RadioButtons and Note text area so it can be read.
- **Update:** Clicking on the update button will save any changes to the current note to the database. Also if any changes are made to a note and an action is attempted which could lose



the changes a warning message is displayed. You can then choose to continue and lose the changes or cancel the action.

- **Delete:** By selecting a record in the column view and then clicking the delete button the currently selected record will be deleted.
- **List:** A list of database records is shown in the column view. Clicking on the Refresh button will load all note titles into the column view.
- **Search:** In the tool box we have a text box where you can enter a search string. Clicking on the search button next to this text box will list all database records that contain this text in a Note Title or Note. If you want to get the complete list of notes back again then you would click the refresh button.

#### 4.4: Adding the code

This is the section where we get our application to do it's real work. The code for this applications is split into two sections. The first is the form window that handles the user interface. We have just created the controls we need for this and the resizing code. Later we shall add code to handle operations created when the user edits notes. The second code section is going to be the interface to the database. Creating this module is what we are going to do next.

Our code to communicate with the database is going to be placed in a Gambas module. Having this database code in one module makes it easier to separate the database specific code from the user interface code. Right click in the Gambas project manager and select **New** from the pop up menu. Then select **Module...** from the sub menu. In the dialog call the module `ModuleDatabase` and leave the check boxes in the default state of unchecked. Click on the **OK** button to create the module.

We first need two variables that are used by a number of procedures. The first is a database connection object. This is only used inside this procedure so it is made `PRIVATE`. The second variable is a Result object. This holds our Notes data as well as being used to update and delete notes. As it is also used by the `FormNotations` class it is made `PUBLIC`.

```
ModuleDatabase.module
```

```
PRIVATE databaseConnection AS NEW Connection
PUBLIC Notes AS Result
```

[Continued Below](#)

Our first procedure opens a database connection. It performs the following actions:

- Open a connection (to the database server only).
- Check if the server connection has a database with the required database name.
- If there is no database with the required database name then create a new database.
- Close the server connection.
- Open a connection to the database.
- Check if the database has a Notes table.
- If there is no Notes table in the database then add this table. Also add a default welcome note with some information about how to use this application.

First we set up the information for the connection to the database server. We use the information passed to the procedure for database type, host name, user name and password. Notice we set the database name property to a blank string. This is because we want to open a connection to the server first and check that the database exists. Having set up the connection we try to open it. This is one of the key points the procedure could produce an error. We then test if the database name we require already exists. If it does not exist then we add this database to the server.

For a SQLite database I found you needed to give the system a short delay to write the newly created database to disk before attempting to open it. This is not required for client/server types of databases like MySQL or PostgreSQL. As it does not do any harm we include it for all types of databases.

If we reach this point then we know there exists a database with the required name. So we set the database name on the connection. (We also set the host name again, this is not strictly necessary, but I think it makes the code a bit more readable.) Then we open the database connection. This is the second key point where we could produce an error.

We then test if the notes table exists in the database. If the table does not exist then we need to add it. We do this by creating a new table object using our database connection. We then add the required fields to this table and state which field is the primary key for the table. In order to write the new table into the database we call the update method on the table object.

The following table shows some information about the fields for our Notes table. The information in the first three columns is required by Gambas. The final columns show what the Gambas data types map to for some database types.

<i>Field Name</i>	<i>Gambas Type</i>	<i>Comment</i>	<i>SQLite 2 Type</i>	<i>MySQL Type</i>
CreateDate	gb.Date	Primary Key	DATETIME	datetime
LastModified	gb.Date		DATETIME	datetime

<i>Field Name</i>	<i>Gambas Type</i>	<i>Comment</i>	<i>SQLite 2 Type</i>	<i>MySQL Type</i>
Title	gb.String	Length 0 for a unlimited string	TEXT	text
Note	gb.String	Length 0 for a unlimited string	TEXT	text
Priority	gb.Integer		INT4	int(11)

We are going to write a procedure to add records to our database. So we may as well use this to create a first record in our database. The content of the note is taken from the welcome text file we [created above](#). As this is a relative file path it would be included in the application when it is compiled.



Gambas version 1 does not understand the `Dconv` function. So lines that include it:

```
... Dconv(Error.Text) ...
```

should just omit the function call:

```
... Error.Text ...
```

This also applies to most of the procedures in `ModuleDatabase` that follow.

ModuleDatabase.module

```
PUBLIC SUB OpenDatabase(DBType AS String, DBHost AS String, DBName
➔ AS String, UserName AS String, UserPassword AS String)
  DIM notesTable AS Table
  DIM errorMessageHeader AS String
  ' Open a connection (to the database server only)
  databaseConnection.Type = Lower(DBType)
  databaseConnection.Host = DBHost
  databaseConnection.Name = ""
  databaseConnection.Login = UserName
  databaseConnection.Password = UserPassword
  databaseConnection.Port = ""
  ' Open the connection
  TRY databaseConnection.Open()
  IF ERROR THEN
    errorMessageHeader = "Could not open database connection " &
➔ DBHost
    Error.Raise(Error.Text)
```

```
END IF
' Check if the server connection has a database with the
' required database name.
IF NOT databaseConnection.Databases.Exist(DBName) THEN
  PRINT "Database not found. Creating new database"
  ' Create new database
  databaseConnection.Databases.Add(DBName)
  ' I found I needed this with a SQLite database
  ' (but not with a MySQL database)
  WAIT 0.5
END IF
' Close the server connection
databaseConnection.Close()
' Open a connection to the database
databaseConnection.Host = DBHost
databaseConnection.Name = DBName
TRY databaseConnection.Open()
IF ERROR THEN
  errorMessageHeader = "Could not open database " & DBName &
  ➔ " on " & DBHost
  Error.Raise(Error.Text)
END IF
' Check if the database has a Notes table
IF NOT databaseConnection.Tables.Exist("Notes") THEN
  PRINT "Database tables not found. Creating new notes table"
  ' Add a Notes table to the database
  notesTable = databaseConnection.Tables.Add("Notes")
  notesTable.Fields.Add("CreateDate", gb.Date)
  notesTable.Fields.Add("LastModified", gb.Date)
  notesTable.Fields.Add("Title", gb.String, 0)
  notesTable.Fields.Add("Note", gb.String, 0)
  notesTable.Fields.Add("Priority", gb.Integer,, 0)
  notesTable.PrimaryKey = ["CreateDate"]
  notesTable.Update()
  ' Add a default welcome record
  AddData("Welcome", File.Load("Welcome.txt"), 0)
```

```
END IF
CATCH
  IF errorMessageHeader = "" THEN
    errorMessageHeader = "Database connection error: " & DBName &
    ➔ " on " & DBHost
  END IF
  Error.Raise("<b>" & errorMessageHeader & "</b><hr>Error:<br>" &
  ➔ DConv(Error.Text))
END
```

### [Continued Below](#)

As you can see most of the above procedure deals with checking we have a valid database and tables. The code to create the connection is relatively small. If you knew you had a valid database then you could cut this procedure down to the following code:

```
databaseConnection.Type = Lower(DBType)
databaseConnection.Host = DBHost
databaseConnection.Name = DBName
databaseConnection.Login = UserName
databaseConnection.Password = UserPassword
databaseConnection.Port = ""
databaseConnection.Open()
```

Also you would need some error checking when you opened the database connection.

When communicating with a database it is important to handle any possible errors. But where is the best place in your application for such code? We want this module to be general and flexible. So here we just format the error message. The Gambas message box can use HTML formatting so we add some information and HTML formatting and pass on the error using the `Error.Raise` method. This means that in the code in the Form class we need to catch these errors and display the error message. There are three possible errors we could have in this procedure:

- When we open a connection (to the database server only).
- When we open a database connection.
- Any other type of error. For example if we do not have permissions to create the database or table.

This module will give a slightly different error message in each case. In the procedures below we adopt a similar approach to handling errors.

The following procedure will close the connection to the database. This procedure is called when we close our application. We place a TRY before the close method as we do not want any errors displayed to the user if the database is already closed. If there is a error closing the database we PRINT as message. This is so you can see errors when developing the application. However the user should not see this message.

ModuleDatabase.module - Continued

```
PUBLIC SUB CloseDatabase()  
    TRY databaseConnection.Close()  
    IF ERROR THEN PRINT "Error closing database"  
END
```

[Continued Below](#)

This next function adds a record to our Notes database table. The values passed to the function are the Note Title, the Note itself and its Priority. For the creation and last modified time we use the current time.

Using the Create method on the connection object we create a Result object that is designed to add records to a database table. We pass the name of the table to the Create method. This returns a Result object that has one empty record in it. The fields in this Result object are the fields in our table. We then fill the fields with the required values for the new record. Then we call the Update method which will add our new record to the database table.

If there are any errors in creating the database record the we format the error message and pass it up to the calling procedure. The function returns a the date and time that the note was created. This date and time is the value of the key field of the newly created note. It allows the calling procedure to find the new note using this key.

ModuleDatabase.module - Continued

```
' Create a record  
PUBLIC FUNCTION AddData(Title AS String, Note AS String, Priority  
    AS Integer) AS String  
    DIM newNote AS Result  
    DIM createTime AS Date  
    createTime = Now
```

```

newNote = databaseConnection.Create("Notes")
newNote["CreateDate"] = createTime
newNote["LastModified"] = createTime
newNote["Title"] = Conv(Title, Desktop.Charset,
↳ databaseConnection.Charset)
newNote["Note"] = Conv(Note, Desktop.Charset,
↳ databaseConnection.Charset)
newNote["Priority"] = Priority
newNote.Update()
RETURN FormatDate(createTime)
CATCH
  Error.Raise("<b>Add database record error</b><hr>Error:<br>" &
↳ DConv(Error.Text))
END

```

### [Continued Below](#)

A Result object is a structure that can store the result of a database query. There are three kinds of Result objects – Create, Read and Write. There are four methods we can use to create a Result object. The functionality of the returned Result object depends on the method used to create it. These methods are summarised in the following table. We have already used the Create method to add a database record - [see above](#). But we also present it here to be complete.

<i>Method</i>	<i>Result Type</i>	<i>Comment</i>
<b>Create</b> (Table AS String) AS Result	Create	Creates a Result object with one record that can be used for adding records to a table.
<b>Find</b> (Table AS String [, Request AS String, Arguments...]) AS Result	Read Only	This method returns data from a single table. Table is the (case insensitive) name of the database table. Request is an optional SQL type WHERE clause used to filter the table rows. Arguments is an optional list of parameters to be substituted in the Request clause. The returned Result object can not be updated.

<i>Method</i>	<i>Result Type</i>	<i>Comment</i>
<b>Edit</b> (Table AS String [, Request AS String, Arguments...]) AS Result	Read/Write	<p>This method returns data from a single table. Table is the (case insensitive) name of the database table. Request is a optional SQL type WHERE clause used to filter the table rows. Arguments is a optional list of parameters to be substituted in the Request clause. The returned Result object is updateable.</p> <p>In the returned Result object records can be updated by moving to a row changing the field values and calling the Update method. Records can be deleted by moving to a row and calling the Delete method. The record is then deleted from the underlying database table.</p> <p>You can not add record to this kind of Result object. See the Create method above.</p>
<b>Exec</b> (Request AS String [, Arguments...]) AS Result	Read Only	<p>Here Request is any SQL query. Arguments is a optional list of parameters that are used in the query. The returned Result object can not be updated.</p> <p>This method can be us send any SQL query to the database. <a href="#">See above</a></p>

We want to be able to edit database records so we use the Edit method to return a Result object. Using the name of our notes table as a parameter to select all the database notes records. If there are any errors in selecting the database records then we format the error message and pass it up to the calling procedure.

ModuleDatabase.module - Continued

```
' Read a table
PUBLIC SUB SelectData()
    Notes = databaseConnection.Edit("Notes")
CATCH
    Error.Raise("<b>Select database records error</b><hr>Error:<br>"
    & DConv(Error.Text))
END
```

[Continued Below](#)



We opened the `Notes` `Result` object with the `Edit` method on the `Connection` object. We passed the `Edit` method the name our `Notes` database table. This means we have a `Result` object that can be edited. Here we want to update a record. The first values passed to the function is the index in the `Result` object of the record we want to update. The first thing this procedure does is to move to this record.

The other values passed to the function are the new values for the `Note Title`, the `Note` itself and its `Priority`. These values are copied to the `Result` object. We also set the `LastModified` field to the current date and time. We do not change the `CreateDate` as this is the key field. When we have set the required values for the tables fields the `Update` method is then called to update the database table. If there are any errors in updating the database record then we format the error message and pass it up to the calling procedure.

ModuleDatabase.module - Continued

```
' Update a record
PUBLIC SUB UpdateData(Row AS Integer, Title AS String, Note AS
↳ String, Priority AS Integer)
    Notes.MoveTo(Row)
    Notes["LastModified"] = Now
    Notes["Title"] = Conv(Title, Desktop.Charset,
↳ databaseConnection.Charset)
    Notes["Note"] = Conv(Note, Desktop.Charset,
↳ databaseConnection.Charset)
    Notes["Priority"] = Priority
    Notes.Update()
CATCH
    Error.Raise("<b>Update database record error</b><hr>Error:<br>"
↳ & Dconv(Error.Text))
END
```

[Continued Below](#)

In order to delete a notes record all we need is the index to the notes record row in the `Result` object. This is passed as a parameter to the delete function. We make sure we are on this row and call the `Result` objects `Delete` method. This deletes the record from the database table. If there are any errors in deleting the database record then we format the error message and pass it up to the calling

procedure.

ModuleDatabase.module - Continued

```
' Delete a record
PUBLIC SUB DeleteData(Row AS Integer)
    Notes.MoveTo(Row)
    Notes.Delete()
CATCH
    Error.Raise("<b>Delete database record error</b><hr>Error:<br>"
➔ & DConv(Error.Text))
END
```

[Continued Below](#)

In the database module we provide a method that converts a date object to a formatted date string. We do this so we can have a common format for the date/time in our application. We always use this method call to format the date. Then if we ever need to change the date format we only have to do this in one place.

ModuleDatabase.module - Continued

```
PUBLIC FUNCTION FormatDate(d AS Date) AS String
    RETURN Format(d, "dd mmm yyyy hh:nn:ss")
END
```

So far we have created a user interface and we have just created a interface to the database. Now we need to bring these two sections together. We need to add the code so that when the user edits a note in the user interface it connects to and updates the database.

The first procedure handles the Form open event. This is run the first time the form is loaded and so is a good place for our set up code. We set the title to the form window to include the application name and version. We have a ColumnView on the left of the form which will show a list of the available notes in the database. Here we configure this ColumnView and set the number of columns and the text for the column headings.

We then call some functions to to display information from the database. First we call the function we [created above](#) to create a database connection. We then call functions to [refresh](#) the notes list in the ColumnView and to [display](#) the currently selected note. Our database procedures can throw errors so we need to catch these and display the error message.

When calling a procedures or function in a module we have to use the format:

```
ModuleName.ProcedureName (...)
```

or

```
object = ModuleName.FunctionName (...)
```

without prefixing the procedures or function with the module name is will not be visible within the calling object. Also only PUBLIC procedures or function in a module can be accessed.



Some older versions of Gambas use `System.Home` to return the users Home directory. So the line that calls our open database method should be changed to the following:

```
ModuleDatabase.OpenDatabase("sqlite", System.Home,  
➔ Application.Name, "", "")
```

```
FormNotations.class
```

```
PRIVATE changesMade AS Boolean
```

```
PUBLIC SUB Form_Open()
```

```
    ME.Title = Application.Name & " - Version: " &
```

```
➔ Application.Version
```

```
    ' ColumnView properties
```

```
    ColumnViewNotes.Columns.Count = 3
```

```
    ColumnViewNotes.Columns[0].Text = "P"
```

```
    ColumnViewNotes.Columns[0].Width = 20
```

```
    ColumnViewNotes.Columns[1].Text = " Title"
```

```
    ColumnViewNotes.Columns[1].Width = 150
```

```
    ColumnViewNotes.Columns[2].Text = " Last Modified"
```

```
    ColumnViewNotes.Columns[2].Width = 150
```

```
    ' Use this connection for a SQLite database
```

```
    ModuleDatabase.OpenDatabase("sqlite", User.Home,
```

```
➔ Application.Name, "", "")
```

```
RefreshNotes()  
IF ColumnViewNotes.Current <> NULL THEN  
    DisplayNote(ColumnViewNotes.Current.Key)  
END IF  
CATCH  
    Message.Warning(ERROR.Text)  
END
```

### [Continued Below](#)

When the form is about to close we need to check if the user has made any changes to the current note that have not been saved yet. As we are going to have to perform this test in a number of places we have written a small [function](#) for this. It will show a message box warning of any unsaved changes. This function returns TRUE if the user wants to cancel the selected action. To cancel the close event on the form we use the STOP EVENT statement. STOP EVENT only cancels the event and prevents any other events handlers for the event being called. It does not exit the current event handler. So we need the RETURN statement to do this.

If there were no changes to the database or the user clicked the **Continue** button when prompted about unsaved changes then we close the database connection and quit the application.

### FormNotations.class - Continued

```
PUBLIC SUB Form_Close()  
    IF TestChangesMade() THEN  
        STOP EVENT  
        RETURN  
    END IF  
    ModuleDatabase.CloseDatabase()  
END
```

### [Continued Below](#)

This is our form resizing code again. We have repeated it here to show its position in the code listing. (If you are using Gambas version 1 your code may look different. [See above.](#))

### FormNotations.class - Continued

```
PUBLIC SUB Form_Resize()  
    HPanelTools.Width = ME.ClientWidth  
    HSplitWindow.Move(0, HPanelTools.Height, ME.ClientWidth,  
↳ ME.ClientHeight - HPanelTools.Height)  
END  
  
PUBLIC SUB HSplitWindow_Resize()  
    TextBoxTitle.Width = PanelNote.ClientWidth  
    TextAreaNote.Resize(PanelNote.ClientWidth,  
↳ PanelNote.ClientHeight - TextAreaNote.Top)  
END
```

### [Continued Below](#)

The next group of procedures handle events when the user clicks on a button in our tool bar. This procedure handles the click event when the user selects the new note button.

First we check which of the note priority radio buttons is selected and set a integer based upon the selection. The priority value 0 is for low priority, the value 1 is for medium priority and the value 2 is for high priority. We then send this value along with the title text and note text to the add note function we [created earlier](#).

Adding the note to the database is very easy. The main problem is finding it again! When the user adds a new note we want the new note to be highlighted in the ColumnView and to be the currently selected note. Our AddData function returns the key of the newly created note. We then call our [refresh note](#) method to update the ColumnView. This should include the new note. We then search the text in the ColumnView **Last Modified** column for the key and make this the current note.

### FormNotations.class - Continued

```
PUBLIC SUB ToolButtonNew_Click()  
    DIM key AS String  
    DIM priority AS Integer  
    IF RadioButtonHigh.Value THEN  
        priority = 2  
    ELSE IF RadioButtonMedium.Value THEN
```

```
    priority = 1
ELSE
    priority = 0
END IF
key = ModuleDatabase.AddData(TextBoxTitle.Text,
↳ TextAreaNote.Text, priority)
RefreshNotes()
' Find the note we just created
IF ColumnViewNotes.MoveFirst() THEN RETURN
REPEAT
    IF key = ColumnViewNotes.Item[2] THEN
        ColumnViewNotes[ColumnViewNotes.Item.Key].Selected = TRUE
        BREAK
    END IF
UNTIL ColumnViewNotes.MoveNext()
' Make the note we created the current note
IF ColumnViewNotes.Current <> NULL THEN
    DisplayNote(ColumnViewNotes.Current.Key)
END IF
CATCH
    Message.Warning(ERROR.Text)
END
```

### [Continued Below](#)

This procedure is to handle when the user clicks on the Delete Note tool bar button. We first check that a note has been selected in the ColumnView to delete. If a note is selected we display a warning message that allows the user to cancel the action. If they select **Yes** to this prompt we call the delete method we [created earlier](#). We then call our [refresh note](#) method to update the ColumnView and then call our [display](#) method to show the default note.

### FormNotations.class - Continued

```
PUBLIC SUB ToolButtonDelete_Click()
    IF ColumnViewNotes.Current <> NULL THEN
        IF Message.Warning("Are you sure you want to delete the
```

```
↳ note? \n\n" & ModuleDatabase.Notes["Title"], "Yes", "Cancel") = 1
↳ THEN
    ModuleDatabase.DeleteData(ColumnViewNotes.Current.Key)
    RefreshNotes()
    IF ColumnViewNotes.Current <> NULL THEN
        DisplayNote(ColumnViewNotes.Current.Key)
    END IF
END IF
ELSE
    Message.Info("No note has been selected to delete")
END IF
CATCH
    Message.Warning(ERROR.Text)
END
```

### [Continued Below](#)

We now handle the event when the Refresh tool bar button is clicked. We want a refresh button because, as we shall see in the next procedure, the user can select just the notes that contain a specific search text. This will refresh the ColumnView and display all notes.

Before we refresh the list we check that there are no unsaved changes to the current note using the function we are going to [create below](#). We then use code similar to that in the [form open](#) event to display the ColumnView list and display the default note.

### FormNotations.class - Continued

```
PUBLIC SUB ToolButtonRefresh_Click()
    IF TestChangesMade() THEN RETURN
    RefreshNotes()
    IF ColumnViewNotes.Current <> NULL THEN
        DisplayNote(ColumnViewNotes.Current.Key)
    END IF
CATCH
    Message.Warning(ERROR.Text)
END
```

### [Continued Below](#)

This procedure handles the event of the user clicking on the Search button. The user will have entered some text in the `TextBoxSearch` next to the search button. The objective of this procedure is then to search through all note titles and the note contents for the search text. And to place all the notes that contain the search text in the `ColumnView`.

We start this procedure by calling our function to test if the current note has changes that have not been saved. We are going to create this function [below](#). Next we test if the user has entered some text to search for in the `TextBoxSearch` control. If there is no text there we display a message to the user.

To perform the search we call the [RefreshNotes](#) procedure sending it the search text as a parameter. We then test if this procedure found any notes with the required search text. If some notes have been found then the `ColumnView`'s current value will be set. If it is set then we call our procedure to [display](#) the current note. If no notes have been found then we clear the controls on the right half of the form window that display the current note and also display a message to the user informing them that the search text has not been found.

#### FormNotations.class - Continued

```
PUBLIC SUB ToolButtonSearch_Click()  
  IF TestChangesMade() THEN RETURN  
  IF TextBoxSearch.Text THEN  
    RefreshNotes(TextBoxSearch.Text)  
    IF ColumnViewNotes.Current <> NULL THEN  
      DisplayNote(ColumnViewNotes.Current.Key)  
    ELSE  
      TextBoxTitle.Text = ""  
      RadioButtonMedium.Value = FALSE  
      RadioButtonHigh.Value = FALSE  
      RadioButtonLow.Value = FALSE  
      TextAreaNote.Text = ""  
      changesMade = FALSE  
      Message.Info("The search text \" & TextBoxSearch.Text &  
↳ "\" has not been found in any note")  
    END IF
```



```
ELSE
    Message.Info("There is no search text to find in notes")
END IF
CATCH
    Message.Warning(ERROR.Text)
END
```

### [Continued Below](#)

This procedure handles the event when the button to update the currently selected note is clicked. We first test if there is a currently selected note and display a message if no note is selected.

Next we find the value to send to the database for the priority from the priority radio buttons. We then save the index key to the current note so we can find it again after we refresh the notes list. We then call our database procedure to update a note. This procedure requires the index key for the note to update along with the information for the note.

We call our procedure to [refresh](#) the notes list in the ColumnView with the new data. This procedure will set the current note to a default note. But here is your be more useful to the user to display the note they have just updated. So we use the index key we have saved to set the current item in the ColumnView. We then call the [display](#) note procedure passing it the index key to display the updated note content.

### FormNotations.class - Continued

```
PUBLIC SUB ToolButtonUpdate_Click()
    DIM priority AS Integer
    DIM noteNumber AS Integer
    IF ColumnViewNotes.Current = NULL THEN
        Message.Info("No note has been selected to update")
        RETURN
    END IF
    ' Get value from radio buttons for priority
    IF RadioButtonHigh.Value THEN
        priority = 2
    ELSE IF RadioButtonMedium.Value THEN
        priority = 1
```

```
ELSE
    priority = 0
END IF
noteNumber = ColumnViewNotes.Current.Key
' Update changed record
ModuleDatabase.UpdateData(noteNumber, TextBoxTitle.Text,
↳ TextAreaNote.Text, priority)
' Refresh the notes list
RefreshNotes()
' Display this note
ColumnViewNotes[noteNumber].Selected = TRUE
DisplayNote(noteNumber)
CATCH
    Message.Warning(ERROR.Text)
END
```

### [Continued Below](#)

The help button is very simple. We load the HTML help file we [created earlier](#) into a message box to display the help.

#### FormNotations.class - Continued

```
PUBLIC SUB ToolButtonHelp_Click()
    Message.Info(File.Load("help.htm"))
END
```

### [Continued Below](#)

This event handler is for when the user clicks on a row in the ColumnView. We then want the note the user clicked on to be displayed. We know the user must have clicked on a valid note as click event is only fired if the ColumnView has some items to select. We check if there have been changes made to the previous note that have not been saved. We are going to create this function [below](#). If the user select to cancel the action we highlight the previous note in the ColumnView. If there are no unsaved changes or the user has selected to continue then we call our [display](#) notes method to show the selected note. The key of the ColumnView view item we have designed to match the index for the note.

FormNotations.class - Continued

```
PUBLIC SUB ColumnViewNotes_Click()  
    ' Check if changes have been made to the previous note  
    IF TestChangesMade() THEN  
        ' User selected to cancel change of note  
        ' Display the previous note  
        ColumnViewNotes[ModuleDatabase.Notes.Index].Selected = TRUE  
    ELSE  
        ' Display the new note  
        DisplayNote(ColumnViewNotes.Current.Key)  
    END IF  
END
```

[Continued Below](#)

This group of event handlers are used to detect if the user has made any changes to the current note. If the text in a TextBox or TextArea is modified the Change event is called. Also the Click event on the RadioButton is used to detect if a RadioButton has been selected. Note that the click event on the RadioButton is also called if the user changed the current RadioButton using the keyboard.

FormNotations.class - Continued

```
PUBLIC SUB TextBoxTitle_Change()  
    changesMade = TRUE  
END  
  
PUBLIC SUB TextAreaNote_Change()  
    changesMade = TRUE  
END  
  
PUBLIC SUB RadioButtonPriority_Click()  
    changesMade = TRUE  
END
```

[Continued Below](#)

We now come to the procedures we have in the `FormNotations` class. We have called all of these procedures in the code above. This first procedure we have called many times above. It is used refresh the `ColumnView` with the current notes from the database.

My first intention was to write separate procedures for refreshing the `ColumnView` notes list and for searching notes for some search text. In some ways this is the more obvious design choice. But if we had used two procedures most of the code in each procedure would have been the same. So it seems better to use a `OPTIONAL` parameter to the `RefreshNotes` procedure as the search text. If this parameter is set then filter out the notes that do not contain this text.

We call our database procedure to [select](#) all the notes in the database. We want the default note to be the first note we find with the highest priority. We use the `firstItem` variable to keep track of the priority of the note we have selected. So we need to initialise `firstItem` to a value lower then any possible priority value in the database. We then convert the `SearchText` variable to lower case so we can perform a case insensitive search for the search text. Next we clear the `ColumnView` of any items that it currently may contain.

Now we come to the main loop of this procedure. Here we loop through all the record that have been returned from the database. We decide if we want to test for the search text by checking if the variable `SearchText` has a value. If `SearchText` has a value then we check if the search text can be found in the notes `Title` field or in the notes `Note` field. If it cannot be found we use the `CONTINUE` statement to skip processing the current record.

We add each item to the `ColumnView` and select the image for the item based upon the priority field. This places the image in the first column of the `ColumnView`. We use the text values "H", "M" and "" (as low) for the priority values so the items can be sorted by priority. (If we used "L" for the low priority items the list would have the low priority items between the high and medium priority items. This would not look very good.)

One important point is using the index from the `Result` object as the key to the `ColumnView` items. This enables us to easily find a record in the `Result` object when we need to update or delete a notes record in the database.

We the add the title of the note to the second column of the `ColumnView`. Then we add the notes last modified date and time to the third column. Here we format the data and time using the [function](#) in our database module. The final check in the loop is to make the default selected item the first item we find with the highest priority.

## FormNotations.class - Continued

```
PRIVATE SUB RefreshNotes(OPTIONAL SearchText AS String)
    DIM firstItem AS Integer
    ModuleDatabase.SelectData()
    firstItem = -1
    SearchText = Lower(SearchText)
    ColumnViewNotes.Clear
    FOR EACH ModuleDatabase.Notes
        IF SearchText THEN
            IF NOT ((InStr(Lower(ModuleDatabase.Notes["Title"]),
↳ SearchText) > 0) OR
                (InStr(Lower(ModuleDatabase.Notes["Note"]), SearchText) >
↳ 0)) THEN
                ' If the search text is not found then go on to the next
↳ record
                CONTINUE
            END IF
        END IF
        IF CInt(ModuleDatabase.Notes["Priority"]) = 1 THEN
            ColumnViewNotes.Add(ModuleDatabase.Notes.Index, "M",
↳ Picture["Images/dialog-information.png"])
        ELSE IF CInt(ModuleDatabase.Notes["Priority"]) > 1 THEN
            ColumnViewNotes.Add(ModuleDatabase.Notes.Index, "H",
↳ Picture["Images/media-record.png"])
        ELSE
            ColumnViewNotes.Add(ModuleDatabase.Notes.Index, "")
        END IF
        ColumnViewNotes[ModuleDatabase.Notes.Index][1] =
↳ ModuleDatabase.Notes["Title"]
        ColumnViewNotes[ModuleDatabase.Notes.Index][2] =
↳ ModuleDatabase.FormatDate(ModuleDatabase.Notes["LastModified"])
        ' Select first item with highest priority
        IF CInt(ModuleDatabase.Notes["Priority"]) > firstItem THEN
            ColumnViewNotes[ModuleDatabase.Notes.Index].Selected = TRUE
            firstItem = CInt(ModuleDatabase.Notes["Priority"])
```

```
END IF
NEXT
END
```

### [Continued Below](#)

This procedure will display a selected note in the details section of the form window on the right side of the form. The procedure expects the index to a row in the Notes Result object to select what's displayed. This why we used this index as the key for items in the ColumnView. We simply need to move to the desired row to get the record we need.

We copy the note title to the title text box. Then we set the selected radio button for the note priority based upon the integer in the Priority field. The note content is copied to the note text area. We place the cursor in the text area at the end of the text and set the focus to the text area. This seems to be the most reasonable position for the user to start from after they have just selected a note.

Finally we set the `changesMade` variable to `FALSE`. To state the obvious: this must be done after we have completed all the changes we need to make to the notes controls. This is because these controls fire changes made events that set this variable to `TRUE`.

### FormNotations.class - Continued

```
PRIVATE SUB DisplayNote(Row AS Integer)
  ModuleDatabase.Notes.MoveTo(Row)
  TextBoxTitle.Text = ModuleDatabase.Notes["Title"]
  IF CInt(ModuleDatabase.Notes["Priority"]) = 1 THEN
    RadioButtonMedium.Value = TRUE
  ELSE IF CInt(ModuleDatabase.Notes["Priority"]) > 1 THEN
    RadioButtonHigh.Value = TRUE
  ELSE
    RadioButtonLow.Value = TRUE
  END IF
  TextAreaNote.Text = ModuleDatabase.Notes["Note"]
  TextAreaNote.Pos = String.Index(TextAreaNote.Text,
  ↳ Len(TextAreaNote.Text))
  TextAreaNote.SetFocus()
  changesMade = FALSE
```

END

### [Continued Below](#)

Our final function tests if the user has made any changes to the current note. When developing the application I found I was writing this kind of logic in a number of places. So it looked like a good idea to write a function for this. This also ensures our application has a common format for handling changes that have not been saved.

We check for changes to the current note by looking at the value of the `changesMade` variable. If there are changes that have not been saved then we display a message box. If the user has clicked the **Continue** button then we return `FALSE` otherwise we return `TRUE`. We have put the function return logic this way round so our message box function behaves the same way as the methods in the `Dialog` class for getting a file name and path.

#### FormNotations.class - Continued

```
PRIVATE FUNCTION TestChangesMade() AS Boolean
    DIM mess AS String
    IF changesMade THEN
        mess = "Changes made to the note\n\n\t" & TextBoxTitle.Text &
↳ "\n\nhave not been saved. Do you want to continue?"
        IF Message.Warning(mess, "Continue", "Cancel") <> 1 THEN
↳ RETURN TRUE
    END IF
    RETURN FALSE
END
```

## **4.5: Running the project**

We have now completed the project so lets run it. You run the project by clicking on the green **Run** button in the project manager window or by pressing the **F5** key. A SQLite database will be created in your user home directory and a table added to the database. Then the welcome note is added to the table. We can now test our application works correctly. There follows some of the tests we might perform. This first set of tests test how we connect to the database.

<i>Database Connection Tests</i>		
<i>Test</i>	<i>Expected Result</i>	<i>Pass/Fail</i>
Run the application with a valid connection and when the database does not exist.	A new database and table should be created. The welcome note should be displayed.	
Run the application again with the valid connection and database.	No changes should be made to the database. The first note found with the highest priority should be displayed.	
Call the database with an invalid connection.	An error message should be displayed.	
After opening the application in the previous step click on all the buttons in the tool bar in turn.	An error message should be displayed,	

There is a forth possible test you could add. This is where we have a valid connection and database but the database table does not exist. For a production application you would want to perform this test. As it is a bit of a complex test new Gambas users might want to skip it.

The next set of test check the adding, reading, updating and deleting of notes. A good way of developing this test list is to use our [CRUD-LS](#) check list as a starting point. Look at each item in the CRUD-LS list and develop tests for when the action succeeds. We also need tests for the all the possible ways each action could fail. The following tests assume we are connected to the database.

<i>Database Create, Read, Update and Delete Tests. Plus List and Search Tests</i>		
<i>Test</i>	<i>Expected Result</i>	<i>Pass/Fail</i>
Change the current note click the <b>Add</b> button.	A new note is added to the database and this becomes the current note.	
Change the current note but make sure the note has the same title as an existing note. Then click the <b>Add</b> button.	A new note is added to the database with the duplicate note title and this becomes the current note.	
Click on the <b>Refresh</b> button. (Make sure any changes to the current note have been saved before you do this.)	The notes list is refreshed from the database. The first highest priority note becomes the current note.	
Click on the <b>Refresh</b> button after changes have been made to a note but have not been saved. In the message box that is displayed click on the <b>Continue</b> button.	The notes list is refreshed from the database. The first highest priority note becomes the current note. Changes to the previous note are lost.	



<i>Database Create, Read, Update and Delete Tests. Plus List and Search Tests</i>		
<i>Test</i>	<i>Expected Result</i>	<i>Pass/Fail</i>
Click on the <b>Refresh</b> button after changes have been made to a note but have not been saved. In the message box that is displayed click on the <b>Cancel</b> button.	The refresh action is cancelled.	
Clear the search text box of any text. Click on the <b>Search</b> button.	A message is displayed stating there is no search text.	
Enter some search text in the search text box. Verify this text can be found in some notes. Click on the <b>Search</b> button. (Make sure any changes to the current note have been saved before you do this.)	Only those notes that contain the search text in the note title or content are shown.	
Enter some search text in the search text box. Verify this text can <i>not</i> be found in some notes. Click on the <b>Search</b> button. (Make sure any changes to the current note have been saved before you do this.)	A message is displayed stating that the search text was not found. No notes are displayed.	
Click on the <b>Search</b> button after changes have been made to a note but have not been saved. In the message box that is displayed click on the <b>Continue</b> button.	Only those notes that contain the search text in the note title or content are shown. Changes to the previous note are lost.	
Click on the <b>Search</b> button after changes have been made to a note but have not been saved. In the message box that is displayed click on the <b>Cancel</b> button.	The search action is cancelled.	
Make some changes to a note and then click on the <b>Update</b> button. Test in turn changes to the title, note and priority.	The current note is updated and the notes list is updated. The updated note remains the current note.	
With a note selected click on the <b>Delete</b> button. Then in the warning message click on the <b>Continue</b> button.	The current note is deleted and the notes list is refreshed.	
With a note selected click on the <b>Delete</b> button. Then in the warning message click on the <b>Cancel</b> button.	The delete action is cancelled.	
Click on the <b>Delete</b> button when either no note is selected or there are no notes in the database table.	A message is displays stating there is no current note to delete.	

Our final batch of tests cover the user interface to our application.

<i>Other Tests</i>		
<i>Test</i>	<i>Expected Result</i>	<i>Pass/Fail</i>
Click on the help button.	The help message should be displayed.	
Click on the heading in the column view of notes.	The notes sort order change should change to the relevant column.	
Test the resizing of the window from each side of the window.	The forms controls should be resized to fit the window.	
Test the resizing of the window from each corner of the window.	The forms controls should be resized to fit the window.	
Resize the window so its width is less than width of all the icons in the tool bar.	Icons that do not fit within the width of the tool bar move on to a second line. The rest of the forms controls are resized appropriately.	
Use the bar between the two half's of the form window to resize each section of the window. (This test does not apply to Gambas Version 1)	The controls in the two half's of the window should be resized correctly.	

#### 4.6: Switching to a MySQL or PostgreSQL Database

We are not going to deal with installing or setting up a MySQL or PostgreSQL database. For information on this you should look at <http://www.mysql.org/> or <http://www.postgresql.org/>.



Performing a internet search for **Installing + MySQL + Linux** finds a large amount of help. Also try **Installing + MySQL + “Your Linux distribution name”**. Similar results can be obtained for PostgreSQL with **Installing + PostgreSQL + Linux**.

Swapping to a one of these databases is very simple. Find the line `Form_Open` event in the `FormNotations` class that looks like this:

```
ModuleDatabase.OpenDatabase("sqlite", User.Home,
↳ Application.Name, "", "")
```

and change it to something like this for a MySQL database:

```
ModuleDatabase.OpenDatabase("mysql", "localhost",
↳ Application.Name, "mysql", "password")
```

Make sure the MySQL user account you select has permissions to create the database and tables the first time this procedure is run. Check on your MySQL installation or with your database

administrator for the MySQL user account details. That all we need to change. So long as you have a valid connection details the program should run and create the database for you.

For a PostgreSQL database we would change the call to something like this:

```
ModuleDatabase.OpenDatabase("postgresql", "localhost",  
↳ Application.Name, "postgres", "password")
```

Also the comment above about account details applies to this type of database.

This example demonstrates the changes required to switch database types. We can see how easy this was. But there are some issues you should consider from a database administrator perspective:

- It is probably not a good idea to hard code the password. It would be better to have a connection dialogue and at least have the password entered by the user. This dialogue would be simple to add in Gambas.
- You need the database user to have permissions to create the database and tables when the program is first run. However is not good practise a leave a normal user with these permissions. This could be a security issue.

This is not really a problem with the SQLite database we first used. SQLite is intended to be a single user database. But as you can see the issues involved when swapping database types have more to do with being a good database administrator than with Gambas related issues.



After changing the database type it would be prudent to rerun the tests in the [Running the project](#) section to ensure your application still works with the new database type.

---

## 5: Appendix 1: Database Commands with Exec and SQL

In this appendix we shall look at converting the Notations application to directly using Structured Query Language (SQL) to communicate with a database. In the main section of this tutorial we avoided using SQL in order to demonstrate more of the Gambas database classes. We also made the design decision to place all the code to communicate with the database inside one module called `ModuleDatabase`. This decision has paid benefits now as all the changes we need to make are confined to this one module.



In the example Notations project that ships with this tutorial there are two extra directories in the project. The sub directory called **ModuleDatabase-SQL** contains a version of the file **ModuleDatabase.module** that follows the code examples in this appendix. Simply copy this file into the main project directory and overwrite the one that already exists there.

In order to test this file you will need to compile the project again. Gambas by default only compiles files that have changed using a project editor. Select **Project** menu in the Gambas project manager. Then select the sub menu option **Compile All**. This will ensure all files are newly compiled and you are running the latest version of the project. You can also click on the Compile all tool bar button in the project manager to achieve the same result.

The sub directory called **ModuleDatabase-Gambas** contains a backup copy of the original file **ModuleDatabase.module**. You can use this file to restore the project to its original format.

The structure of the commands we need is fairly straightforward. We only need two Gambas database objects, a connection object and a result object. The `Exec` method on the connection object returns a read only result object. So to get records from the database we construct our SQL statement, pass this to the `Exec` method, and then we can read the returned data from the result object:

```
sql = "SQL select statement"
result = connection.Exec(sql)
```

You also can use the `Exec` method to send any valid SQL statements to your target database. So this method can also be used when we add, update or delete records from the database.

```
sql = "Create SQL update, insert or delete statement(s)"
connection.Exec(sql)
```

Of course with these method you still need to catch any errors returned. We are not going to deal with how to use SQL in this tutorial. If you need information on SQL a good place to start is the web sites for the database vendors listed in the [Gambas Resources](#) section.

There are two important issues to consider when constructing the string for the SQL statement. The first is we need to format the date/time in a way that is unambiguous and understood by the database. This fragment shows how we format the current date/time:

```
"" & Format(Now, "yyyy-mm-dd hh:nn:ss") & ""
```

This will produce a string something like '2006-05-13 19:38:34' at the time of writing this sentence (on 13 May 2006 at 19:38:34).

In our SQL statements we have used text strings that are surrounded by single quotes. This means that if the user enters a single quote in their title or note entry then the INSERT or UPDATE SQL statement would fail. This would be confusing to the user as they would just see an bad SQL query error message and not know how to correct the problem. Hence we have adopted the (admittedly crude) approach of replacing any single quotes with double quotes when constructing our SQL query. (You can include single quotes within a text string in the version of this application using Gambas objects to add and update records.)

This is a listing of the changes we need to make to the ModuleDatabase code to use the SQL method for communications to the database.



After entering this listing you should rerun the all the tests in the [Running the project](#) section to ensure your application still works with this different method of communicating with the database.

Also if you change the database type, as described in, [Switching to a MySQL or PostgreSQL Database](#) you should be very thorough in rerunning any testing. You need to repeat all your testing for each type of database you intend to support. You need to do more retesting than it you used the Gambas method of communicating with the database.



In the following listing you need to make a few small changes for Gambas version 1. The lines that have the format:

```
sql &= "xxx"
```

need to be changed to:

```
sql = sql & "xxx"
```

as Gambas version 1 does not understand the &= operator. Also Gambas version 1 does not understand the Dconv function. So lines that include it:

```
... Dconv(Error.Text) ...
```

should just omit the function call:

```
... Error.Text ...
```

ModuleDatabase.module

```
PRIVATE databaseConnection AS NEW Connection
PUBLIC Notes AS Result

' This procedure will open a SQLite database connection.
' If the database does not exist it will be created.
' If the tables do not exist they will be created.
PUBLIC SUB OpenDatabase(DBType AS String, DBHost AS String, DBName
↳ AS String, UserName AS String, UserPassword AS String)
    DIM sql AS String
    DIM errorMessageHeader AS String
    ' Open a connection (to the database server only)
    databaseConnection.Type = Lower(DBType)
    databaseConnection.Host = DBHost
    databaseConnection.Name = ""
    databaseConnection.Login = UserName
    databaseConnection.Password = UserPassword
    databaseConnection.Port = ""
    ' Open the connection
    TRY databaseConnection.Open()
    IF ERROR THEN
        errorMessageHeader = "Could not open database connection " &
↳ DBHost
        Error.Raise(Error.Text)
    END IF
    ' Check if the server connection has a database with the
    ' required database name.
    IF NOT databaseConnection.Databases.Exist(DBName) THEN
        PRINT "Database not found. Creating new database"
        ' Create a new database
        databaseConnection.Databases.Add(DBName)
        ' I found I needed this with a SQLite database
        ' (but not with a MySQL database)
        WAIT 0.5
    END IF
```

```
' Close the server connection
databaseConnection.Close()
' Open a connection to the database
databaseConnection.Host = DBHost
databaseConnection.Name = DBName
TRY databaseConnection.Open()
IF ERROR THEN
    errorMessageHeader = "Could not open database " & DBName &
➔ " on " & DBHost
    Error.Raise(Error.Text)
END IF
' Check if the database has a Notes table
IF NOT databaseConnection.Tables.Exist("Notes") THEN
    PRINT "Database tables not found. Creating new notes table"
    ' Add a Notes table to the database
    sql = "CREATE TABLE 'Notes' ( "
    sql &= "CreateDate DATETIME NOT NULL, "
    sql &= "LastModified DATETIME, title TEXT, "
    sql &= "Note TEXT, "
    sql &= "Priority INTEGER NOT NULL DEFAULT 0, "
    sql &= "PRIMARY KEY (CreateDate) );"
    databaseConnection.Exec(sql)
    ' Add a default welcome record
    AddData("Welcome", File.Load("Welcome.txt"), 0)
END IF
CATCH
    IF errorMessageHeader = "" THEN
        errorMessageHeader = "Database connection error: " & DBName &
➔ " on " & DBHost
    END IF
    Error.Raise("<b>" & errorMessageHeader & "</b><hr>Error:<br>" &
➔ DConv(Error.Text))
END

PUBLIC SUB CloseDatabase()
    TRY databaseConnection.Close()
```

```
IF ERROR THEN PRINT "Error closing database"
END

' Create a record
PUBLIC FUNCTION AddData(Title AS String, Note AS String, Priority
↳ AS Integer) AS String
  DIM sql AS String
  DIM createTime AS Date
  createTime = Now
  Title = Replace(Title, "'", "\"")
  Note = Replace(Note, "'", "\"")
  sql = "INSERT INTO Notes "
  sql &= "(CreateDate, LastModified, Title, Note, Priority) "
  sql &= "VALUES ("
  sql &= "'" & Format(createTime, "yyyy-mm-dd hh:nn:ss") & "', "
↳ ' CreateDate
  sql &= "'" & Format(createTime, "yyyy-mm-dd hh:nn:ss") & "', "
↳ 'LastModified
  sql &= "'" & Conv(Title, Desktop.Charset,
↳ databaseConnection.Charset) & "', "
  sql &= "'" & Conv(Note, Desktop.Charset,
↳ databaseConnection.Charset) & "', "
  sql &= Priority & ");"
  databaseConnection.Exec(sql)
  RETURN FormatDate(createTime)
CATCH
  Error.Raise("<b>Add database record error</b><hr>Error:<br>" &
↳ DConv(Error.Text))
END

' Read a table
PUBLIC SUB SelectData()
  DIM sql AS String
  sql = "SELECT * FROM Notes;"
  Notes = databaseConnection.Exec(sql)
CATCH
```



```
Error.Raise("<b>Select database records error</b><hr>Error:<br>"
↳ & DConv(Error.Text))
END

' Update a record
PUBLIC SUB UpdateData(Row AS Integer, Title AS String, Note AS
↳ String, Priority AS Integer)
  DIM sql AS String
  DIM createDate AS Date
  Title = Replace(Title, "'", "\"")
  Note = Replace(Note, "'", "\"")
  Notes.MoveTo(Row)
  sql = "UPDATE Notes "
  sql &= "SET LastModified = '" & Format(Now,
↳ "yyyy-mm-dd hh:nn:ss") & "', "
  sql &= "Title = '" & Conv(Title, Desktop.Charset,
↳ databaseConnection.Charset) & "', "
  sql &= "Note = '" & Conv(Note, Desktop.Charset,
↳ databaseConnection.Charset) & "', "
  sql &= "Priority = " & Priority & " "
  createDate = Notes["CreateDate"]
  sql &= "WHERE CreateDate = '" & Format(createDate,
↳ "yyyy-mm-dd hh:nn:ss") & "';"
  databaseConnection.Exec(sql)
CATCH
  Error.Raise("<b>Update database record error</b><hr>Error:<br>"
↳ & DConv(Error.Text))
END

' Delete a record
PUBLIC SUB DeleteData(Row AS Integer)
  DIM sql AS String
  DIM createDate AS Date
  Notes.MoveTo(Row)
  createDate = Notes["CreateDate"]
  sql = "DELETE FROM Notes "
```

```
    sql &= "WHERE CreateDate = '" & Format(createDate,  
↳ "yyyy-mm-dd hh:nn:ss") & "';"  
    databaseConnection.Exec(sql)  
CATCH  
    Error.Raise("<b>Delete database record error</b><hr>Error:<br>"  
↳ & DConv(Error.Text))  
END  
  
PUBLIC FUNCTION FormatDate(d AS Date) AS String  
    RETURN Format(d, "dd mmm yyyy hh:nn:ss")  
END
```

---