VALENTINA
for REALbasic Tutorial

Paradigma (www.paradigmasoft.com)

Acknowledgments:

**Andy Bachorski , Andy Fuchs, Bill Mounce, Brian Blood,
Craig A. Berry, David A. Bayly, Frank J. Schima, Guillermo Zaballa,
Hideaki Iimori, John Roberts, Lynn Fredricks, Paul Shaap, Robert Brenstein.**

Contents

Introduction

Valentina for REALbasic can be thought of as your own database-engine which you build into your application developed with REALbasic. This is a slightly different method than the database architecture of REALbasic Pro, which suggests that you consider your application as a client of a foreign remote database.

Like C++ and other high level languages, REALbasic allows you to develop standard Macintosh applications meeting a variety of needs. To incorporate a database into your application, you will need to choose between a static database structure or a dynamic database structure. While your development may include facets of each type, the actual programming is considerably different for each type of structure. A brief discussion of these is included below, but rest assured that Valentina for REALbasic provides the tools for bothstatic and dynamic database structures.

Applications with a Static Database Structure

Applications with a static database structure have a fixed number of named tables and each table has a fixed number of named fields. Each field is assigned a type which never varies. Because you know the names of the tables and fields, you can easily incorporate these as program variables by including them as properties of the database object and then using them directly in your code.

Applications with a Dynamic Database Structure

Applications with a dynamic database structure are usually called Database Management Systems (“DBMS”). These allow the user to create, delete and modify database tables and the associated fields. A DBMS is designed to work on an arbitrary database. To accomplish this, a DBMS will use an array of tables and each table will have an array of fields. The application must provide a user interface for issuing the commands necessary for managing the database structure.

Most of your applications will have a static database structure. For example, an email application has a predefined database structure to contain information which is consistent with the storing and retrieving of email messages. Other examples could include address books, library catalogs and inventory control programs.

Architecture of Valentina for REALbasic

REALbasic is an excellent product, which offers an object-oriented programming environment very similar to the C++ language. Valentina for REALbasic makes extensive use of the object-oriented features of REALbasic.

For applications with a static database structure, we highly recommend that you define the structure of a database via a set of classes derived from the database classes of Valentina. This allows you to work with a database as with normal properties. Besides, this offers a great way to implement object-oriented databases.

Installing Valentina for REALbasic

To install the V4RB plug-in, drag it to the folder "plugins" in the REALbasic folder.
The minimal requirements of Valentina for REALbasic:
• Macintosh with any PPC processor
• MacOS 7.5 or greater (OS 8.6 recommended)
• REALbasic Standard Edition or greater.

Working with Valentina you should always keep keep issues of RAM in mind.
Valentina uses caching to improve database operations.
Besides resolving queries Valentina uses free RAM for additional allocations.

Getting Started with the Examples

In Examples 1-4, we consider the technique of using classes to define a database structure . In examples 5-6, we consider how to implement an application with a dynamic database structure without using classes.

Since the V4RB implementation of the Valentina kernel is done with classes, you can see dramatic speed improvements over databases which use a convertion to string method.

To work with the examples, you need to assign REALbasic between 15 to 20 MB of RAM, using the GetInfo dialog. Examples 1 to 3 allocate 3 MB to the Valentina cache; Examples 4 to 6 allocate 7 MB to the Valentina cache.

Initializing the Valentina Kernel

To begin your own project, you need to create an application sub-class in it. Then in the Open and Close methods of the application sub-class write the following code:

```
TestApplication.Open:
Sub Open()
	dim Err as integer

	Err = ValentinaDebugON( 1 )

	 // init database kernel of Valentina
	Err = ValentinaInit( 3 * 1024 *1024, "", "" )
	if Err <> 0 then
		MsgBox "Valentina was not initialized !!! Error: " + Str(Err)
	end if
End Sub

TestApplication.Close:
Sub Close()
	dim i as integer

	For i=0 to Ubound(windows)
		If Windows(i)<> nil Then
			Windows(i).Close
		End If
	Next

	// now we can shutdown Valentina because all database
	// are closed and their objects are destroyed.
	ValentinaShutDown

End Sub
```

This code initializes the kernel of Valentina at the start of the application and exits the kernel on quit. Valentina can handle multiple open databases, in all our examples we will use multiple databases, each attached to its own window. Therefore, we need to establish an array of all open windows as properties of the application. This is so that on closing the application, first all database windows are closed, then the Valentina kernel is shut down.

If your application will only have one open database at a time, then it is not problematic to make it a global database. In this case, you don't have to be concerned with arrays of windows, and your Close method could be as simple as:

```
Sub Close()
	 ValentinaShutDown
End Sub
```

Example 1 (Single Table Database)

In the first example, we will build an application with a static database structure which has one table in the database. This table will have 11 fields of each type which Valentina supports except BLOB fields: boolean, byte, ushort, short, long, ulong, float, double, date, time, string.

We will then write methods which will add 100'000 records to our table. Next, we will make a window which will display records in the EditFields and have navigation buttons. Finally, we will make a query window which allows you to write SQL queries, execute them, and view the results.

Define database classes

1) We begin the process by adding a new class "MyDatabase" to the project which will be a child of the VDataBase class. This class will manage our database.

2) Although we want to have only one Table (a Base Object in the terminology of Valentina) in the Database, we need to create yet another class, which will be a child of VBaseObject. Let's name it as "boPerson" (prefix "bo" just means base object).

3) Since the database must know its Tables, let's add the following property to the class "MyDatabase":

```
mPerson    as boPerson
```

4) We want to create an instance of the class "boPerson" when we create an instance of the DataBase. To implement this, we need to add a constructor to the class "MyDataBase", i.e. a method with the same name as the name of the class. In the constructor of the database we will create an object of Person:

```
Sub MyDatabase
    mPerson = new boPerson
End Sub
```

5) Now we need to define fields of the Table.
For this, add the following 11 properties to the class "boPerson" (see example):

```
BoolFld      as VBoolean
ByteFld      as VByte
ShortFld     as VShort
UShortFld    as VUshort
LongFld      as VLong
ULongFld     as VULong
FloatFld     as VFLoat
DoubleFld    as VDouble
StringFld    as VString
DateFld      as VDate
TimeFld      as VTime
```

6) For BaseObject “boPerson” we also need to define a constructor where we will create fields of the Table:

```
Sub boPerson()
        // set name of the Table (BaseObject)
        name = "Person”

        // Make fields of the BaseObject (Table)
        BoolFld = new VBoolean( “bool_fld” )
        ByteFld = new VByte( "byte_fld”, kV_Nullable )

        ShortFld = new VShort( "short_fld" )
        UShortFld = new VUShort( "ushort_fld” )

        LongFld = new VLong( "long_fld” )
        LongFld = new VULong( "ulong_fld” )

        FloatFld = new VFloat( "float_fld” )
        DoubleFld = new VDouble( "double_fld” )

        StringFld = new VString( "string_fld”,  30 )

        DateFld = new VDate( "date_fld” )
        TimeFld = new VTime( "time_fld” )
End Sub
```

The name surrounded by the quotation marks is the name for the field inside of the database.

That is all! We have defined the structure of the database via classes!

Let’s repeat the steps:
1. Create a new class MyDatabase as a child to VDatabase class.
2. Create a new class boPerson as a child to VBaseObject class
3. Add the property mPerson to the MyDatabase class.
4. Add a constructor to MyDatabase class.
5. Add fields of table as properties to boPerson class.
6. Add a constructor to boPerson

NOTES:
• The field ByteFld we have specified to be nullable.
• We need only one line of code to define one field.
• The order of fields in the constructor defines the order of fields in the Table.
• For Valentina, order of fields in the table is not important at all, because each field is stored in the separate logical file.
• Constant kV_Nullable is defined in “ValentinaUtilities” module. You need drag this module from folder V4RB into your project.

Don’t confuse an object of class Database with the database on disk. When you create a new object of class Database, you create an object in RAM. After that you can create a new database on disk or open an existing one.

You might ask, where and when must we create an object of the database? Where must a reference to a database object be stored?

The most common way is to relate a database object with a window, i.e. a window managing the database content which is displayed in this window. Of course, in your own application you can have a group of different windows for each database, but in this case there must be one “main database window” also.

Let’s do that in our example.
In the class Window1 (which must already be in your project) add the following property:

```
mDataBase        as MyDataBase
```

On close of Window1 we need to destroy the DataBase object in RAM:

```
Window1.Close:
Sub Close()
    // Close disk files
    mDatabase.Close

    // and Destroy database object in RAM
    mDatabase = nil
End Sub
```

Creating/Opening a Database

To implement a standard MacOS document-oriented application we need to write the class menu handlers “New” and “Open” in the Application.

To implement a standard MacOS document-oriented application, we need to write the class menu handlers “New” and “Open” in the Application.

On “FileNew” we ask the user where the new disk file must be located and create a new database on the disk. Now we ask Valentina to keep the database in 1 disk file with 32 KB segments.

```
TestApplication.FileNew:
Sub FileNew()
	dim f as FolderItem

	f = GetSaveFolderItem( “VALA”, “DataBase1” )
	if( f <> nil )
		mDataBase.Create( f, 1, 32*1024 )
	end if
End Sub
```

On “FileOpen” we ask the user where the existing database is located and open it.

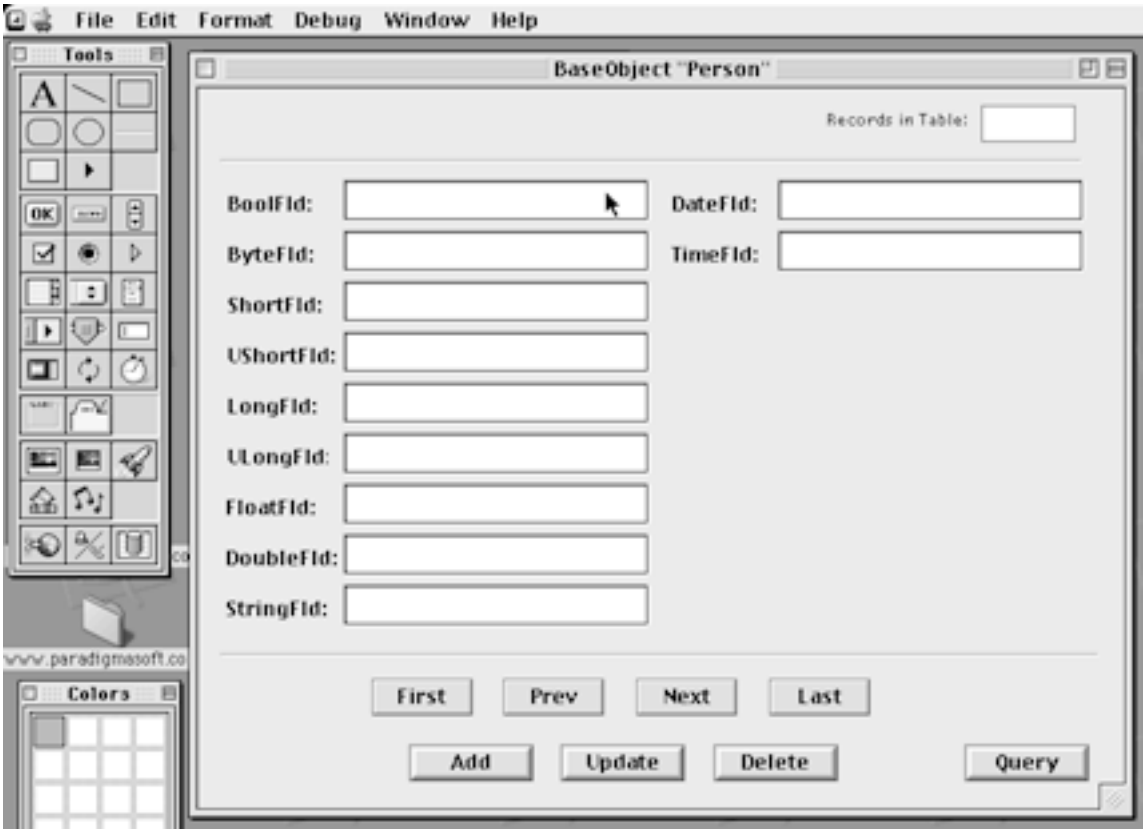
```
TestApplication.FileNew:
Sub FileNew()
	dim f as FolderItem

	f = GetOpenFolderItem( “VALA”, “DataBase1” )
	if( f <> nil )
		mDataBase.Open( f )
	end if
End Sub
```

We also need to enable the menu items “New...” and “Open...”

```
TestApplication.EnableMenuItems:
Sub EnableMenuItems()
	FileNew.enabled = true
	FileOpen.enabled = true
End Sub
```

Edit fields in the database window

To display records of a BaseObject, we need to add to Window1 of our project 11 EditFields, one for each field of the Table.



Now we need to create 2 methods: one will move data from a database record to the EditFields to display them, and the other will collect data from the EditFields and store them in the fields of the database. See the next page.

```
Sub PopulatePanes()
	dim res as Boolean

	if( mDataBase = nil ) then
		return
	end if

	if( mDataBase.mPerson.GetRecID = 0 ) then
		res  = mDataBase.mPerson.FirstRecord
	end if

	TotalRecs.Text    = Str( mDataBase.mPerson.RecordCount )

	EditField1.Text   = mDataBase.mPerson.BoolFld.GetString
	EditField2.Text   = Str( mDataBase.mPerson.ByteFld.Value )
	EditField3.Text   = Str( mDataBase.mPerson.ShortFld.Value )
	EditField4.Text   = Str( mDataBase.mPerson.UShortFld.Value )
	EditField5.Text   = Str( mDataBase.mPerson.LongFld.Value )
	EditField6.Text   = Str( mDataBase.mPerson.ULongFld.Value )
	EditField7.Text   = Str( mDataBase.mPerson.FloatFld.Value )
	EditField8.Text   = Str( mDataBase.mPerson.DoubleFld.Value )
	EditField9.Text   = mDataBase.mPerson.StringFld.Value
	EditField10.Text  = mDataBase.mPerson.DateFld.GetString
	EditField11.Text  = mDataBase.mPerson.TimeFld.GetString
End Sub

Window1.CollectPanes:
Sub CollectPanes()

	if( mDataBase = nil ) then
		return
	end if

	mDataBase.mPerson.BoolFld.SetString( EditField1.Text )
	mDataBase.mPerson.ByteFld.Value       = Val(EditField2.Text)
	mDataBase.mPerson.ShortFld.Value      = Val(EditField3.Text)
	mDataBase.mPerson.UShortFld.Value     = Val(EditField4.Text)
	mDataBase.mPerson.LongFld.Value       = Val(EditField5.Text)
	mDataBase.mPerson.ULongFld.Value      = Val(EditField6.Text)
	mDataBase.mPerson.FloatFld.Value      = Val(EditField7.Text)
	mDataBase.mPerson.DoubleFld.Value     = Val(EditField8.Text)
	mDataBase.mPerson.StringFld.Value     = EditField9.Text
	mDataBase.mPerson.DateFld.SetString( EditField10.Text )
	mDataBase.mPerson.TimeFld.SetString( EditField11.Text )
End Sub
```

In this example, we show 2 different ways of performing the same function – using the field property Value and using the methods GetString/SetString. The use of either method is up to the developer.

Navigation buttons in the database window

To navigate through the records of the BaseObject, we will add several buttons to the database window. Their Action methods are below:

```
Window1.BevelButton1.Action:                    // button “First”
Sub Action()
        dim res as Boolean
        res = mDataBase.mPerson.FirstRecord
        PopulatePanes
End Sub

Window1.BevelButton2.Action:                    // button “Prev”
Sub Action()
        dim res as boolean
        res = mDataBase.mPerson.PrevRecord
        PopulatePanes
End Sub

Window1.BevelButton3.Action:                    // button “Next”
Sub Action()
        dim res as boolean
        res = mDataBase.mPerson.NextRecord
        PopulatePanes
End Sub

Window1.BevelButton4.Action:                    // button “Last”
Sub Action()
        dim res as Boolean
        res = mDataBase.mPerson.LastRecord
        PopulatePanes
End Sub

Window1.BevelButton5.Action:                    // button “Add”
Sub Action()
        mDataBase.mPerson.SetBlank      // Clear memory buffer
        CollectPanes                    // Read data from panes to the fields

        mDataBase.mPerson.AddRecord     // Add new record to the table
        PopulatePanes                   // Update panes (records number)
End Sub

Window1.BevelButton6.Action:                    // button “Update”
Sub Action()
        CollectPanes                    // Read data from panes to the fields
        mDataBase.mPerson.UpdateRecord  // Update current record of the table
End Sub

Window1.BevelButton7.Action:                    // button “Delete”
Sub Action()
        mDataBase.mPerson.DeleteRecord
        PopulatePanes
End Sub
```

Adding records to a database

To test the speed of Valentina, we should have a database with thousands of records. Let's write a method which will add records to table boPerson.

```
boPerson.TestAddRecords:
Sub TestAddRecords(N as integer)
	dim i as integer

	for i = 1 to N
		// We want to add a new record, it is good practice to clear the memory
		// buffer of the BaseObject first.
		SetBlank

		BoolFld.Value = (i Mod 2) = 1

		if( i Mod 23 <> 0 ) then
			ByteFld.Value = i Mod 256		// truncate to range to 0.255
		else
			// do nothing, so field of this record will have NULL value.
		end if

		ShortFld.Value	= - i			// don't truncate.
		UShortFld.Value = i			// because it will happen automatically.


		LongFld.Value	= -2000000 * i
		ULongFld.Value	= 2000000 * i

		FloatFld.Value = 3.1415 * i
		DoubleFld.Value = 3.1415 * i * 10000

		if( (i mod 2) = 1 ) then
			StringFld.Value = "line " + Str(i)
		else
			StringFld.Value = "other line " + Str(i)
		end if

		DateFld.Set( (i mod 2500) + 1, (i mod 12) + 1, (i mod 31) + 1 )
		TimeFld.Set( i mod 12, i mod 60,	i mod 60 )

		AddRecord
	next

	// Because we have added many records, we should flush this BaseObject (Table)
	// from cache to disk.
	Flush

End Sub
```

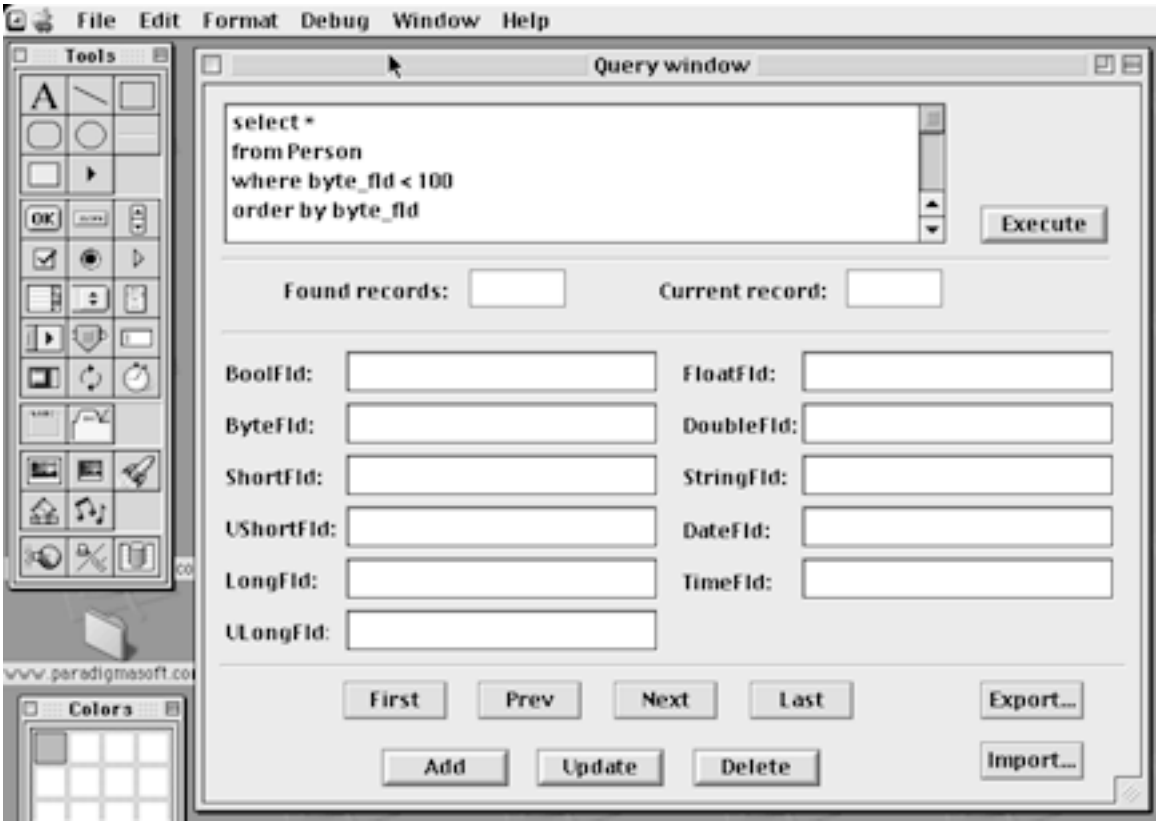
This method will be called from the Window1 menu event handler.

Query window

Let's make another window in our project which will be used to perform SQL queries and for looping through records of a cursor. The Query window must have this property:

```
mCursor     as VCursor
```

This window is very similar to Window1 – the same 11 EditFields, and the same navigation buttons. We can copy all these controls from Window1 by drag and drop.



We will add another multi-line scrollable EditField "SQLString" for entering SQL queries. We will also add the button "Execute" with the following Action:

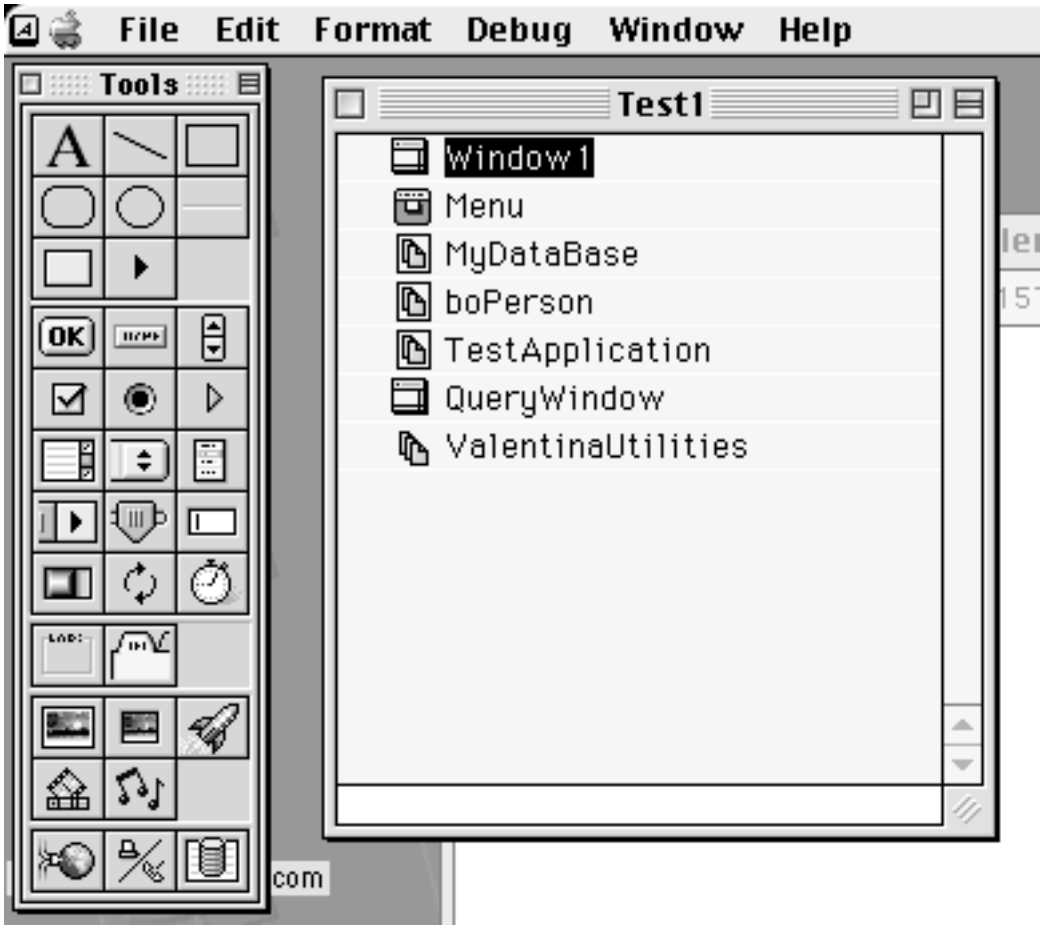
```
QueryWindow.ButtExecute.Action:
Sub Action()
        mCursor = nil                   // destroy results of prev query

        mCursor = mCallerWindow.mDataBase.SQLselect( SQLString.Text )

        mCursor.CurrentPosition = 1
        PopulatePanes
End Sub
```

Important: while navigation buttons of the main Window use methods of the VBaseObject class, the corresponding buttons of the QueryWindow use the VCursor class:

```
QueryWindow.BevelButton1.Action:                    // button “First”
Sub Action()
        mCursor.CurrentPosition = 1
        PopulatePanes
End Sub

QueryWindow.BevelButton2.Action:                    // button “Prev”
Sub Action()
        if( mCursor.CurrentPosition > 1 ) then
                mCursor.CurrentPosition = mCursor.CurrentPosition - 1
                PopulatePanes
        end if
End Sub

QueryWindow.BevelButton3.Action:                    // button “Next”
Sub Action()
        if( mCursor.CurrentPosition < mCursor.RecordCount ) then
                mCursor.CurrentPosition = mCursor.CurrentPosition + 1
                PopulatePanes
        end if
End Sub

QueryWindow.BevelButton4.Action:                    // button “Last”
Sub Action()
        mCursor.CurrentPosition = mCursor.RecordCount
        PopulatePanes
End Sub
```

Working with example 1

1) Run example 1



2) Select File - New from the menu bar
to create a new database on disk.

3) Select Test - “Add 100’000 records” from the menu bar
This command will add 100’000 records to the database.

4) In Window1, you can now see the resulting records.
Try to navigate through the records using the “First”, “Last”, “Prev”, and “Next” buttons.

5) Click the “Query” button to open a QueryWindow.

6) In the top edit field, there is already an example SQL query. Click the “Execute” button. In the edit fields, the found records will be displayed. You can navigate through selected records using the “First”, “Last”, “Prev”, and “Next” buttons.

7) Now try entering some of your own SQL queries.

Example 2 (BLOB)

The second example demonstrates the use of BLOB fields in Valentina. The first BLOB we will be used to store pictures, the second BLOB to store a large amount of text.

The second example is based on the first. So we can duplicate the project of example 1 and rename it.

To class boPerson we add 2 properties:

```
PictureFld      as VBLOB
TextFld         as VText
```

In the constructor of boPerson we add 2 lines:

```
PictFld = new VBLOB( “pict_fld”,  30 * 1024 )
TextFld = new VText( “text_fld”, 2 * 1024 )
```

To Window1 we must add the control ImageWell to display a picture and another multi-line EditField to display the text.

Change method boPerson.AddTestRecords to fill the 2 new fields:

```
select case ( i Mod 3)
        case 0
                pic = picture1
        case 1
                pic = picture2
        case 2
                pic = picture3
end select

PictFld.SetPicture( pic )
TextFld.Value = "This is big, very big notes text, several KB :-)”
```

Here picture1, picture2 and picture3 are some arbitrary pictures.

That is all. The only thing to note -- BLOBs are slows down speed of adding of new records (but not speed of searching and sorting) so we will add not too many records in this test.

Example 3 (2 tables, One to Many relation)

In this example, we will demonstrate working with 2 Tables in a database.
This example is based on the project of example 1.

We want to have in the database 2 Tables – “Person” and “Task” related as One to Many [1:M], i.e. one person can have zero, one or several tasks. But each task belongs to one and only one Person.

We already have the “Person” Table in our database. To add the “Task” table

1) Add to the project a new class “boTask” as a child of class VBaseObject

2) Add a property to class MyDataBase – a reference to the second table:

```
mTask        as boTask
```

3) Add to class boTask the following properties:

```
mPersonPtr   as VObjectPtr
mName        as VString
mStartTime   as VTime
mFinishTime  as VTime
mStatus      as VByte
```

4) Make a constructor for class “boTask” where we define the name of the table and make objects of the fields using the “new” operator:

```
boTask.boTask:
Sub boTask(inDataBase as MyDataBase)
    // Set name of table
    name = "task”

    // Make fields of table
    mPersonPtr   = new VObjectPtr( "person_ptr”, inDataBase.mPerson, kV_SetCascade )
    mName        = new VString( "name”, 20 )
    mStartTime   = new VTime( "start_time” )
    mFinishTime = new VTime( "finish_time” )
    mStatus      = new VByte( "status” )
End Sub
```

Important: the field mPersonPtr requires a reference to the child BaseObject. To access an instance of the boPerson class, we need to pass an instance of MyDataBase as a parameter to the constructor.

We have specified the parameter DeletionControl of field mPersonPtr as kV_SetCascade. This means that if a record of the Table “Person” will be deleted, then all related records in the Table “Task” will be deleted too. In other words, we have specified the RULE: “if there is no Person – there will be no tasks for that person”.

5) Add creation of boTask object to the constructor of class MyDataBase:

```
Sub MyDatabase()
        mPerson = new boPerson
        mTask = new boTask( self )
End Sub
```

Why do we pass an instance of MyDataBase to the constructor of boTask but not an instance of boPerson? Because if our database has many tables in the database, then you will be able access any of them via the instance of the DataBase: inDataBase.mPerson, inDataBase.OtherTable, etc.

6) Update methods for adding records

We need to change our methods which add records to the database, because now we need to fill 2 tables with data.

The method boPerson.TestAddRecords does not need to be modified, but we now need to fill the second table. Also, we need to relate records in the Table “Task” to records in the Table “Person”. For example, we will add 4 Tasks for each person.

Add method TestAddRecords() to the MyDataBase class as follows:

```
Sub TestAddRecords()
        dim res as Boolean
        mPerson.TestAddRecords(100000)

        // Now we will add records to the second table
        res = mPerson.FirstRecord

        // This loop will add 400”000 records to the table "Task”
        do
                mTask.TestAddRecords( mPerson.GetRecID )
        loop until mPerson.NextRecord = false

        Flush
End Sub
```

Add method TestAddRecords() to the boTask class as follows:

```
Sub TestAddRecords( PersonID as integer )
        dim i as integer
        for i = 1 to 4
                SetBlank

                mPersonPtr.value = PersonID        // HERE WE RELATE RECORDS !!!
                mName.value = “task” + str(i)
                mStartTime.Set( i, 0, 0 )
                mFinishTime.Set( i + 6, 20, 40 )
                mStatus.value = i
                AddRecord
        next
End Sub
```

Working with Example 3

1) Run the project of example 3

2) Make a new database on disk using New from the File menu.

3) Choose “Add 500’000 records” from the Test menu.

4) Go to the query window.

Now you can try a SQL query for both tables:

```
SELECT *
FROM Person, Task
WHERE person.id = task.person_ptr AND byte_fld < 10 AND status = 1
ORDER BY string_fld DESC
```

Important: Valentina is smart enough to resolve the same query without an explicit specification of a LINK condition

```
SELECT *
FROM Person, Task
WHERE byte_fld < 10 AND status = 1
ORDER BY string_fld DESC
```

You can try any other queries.

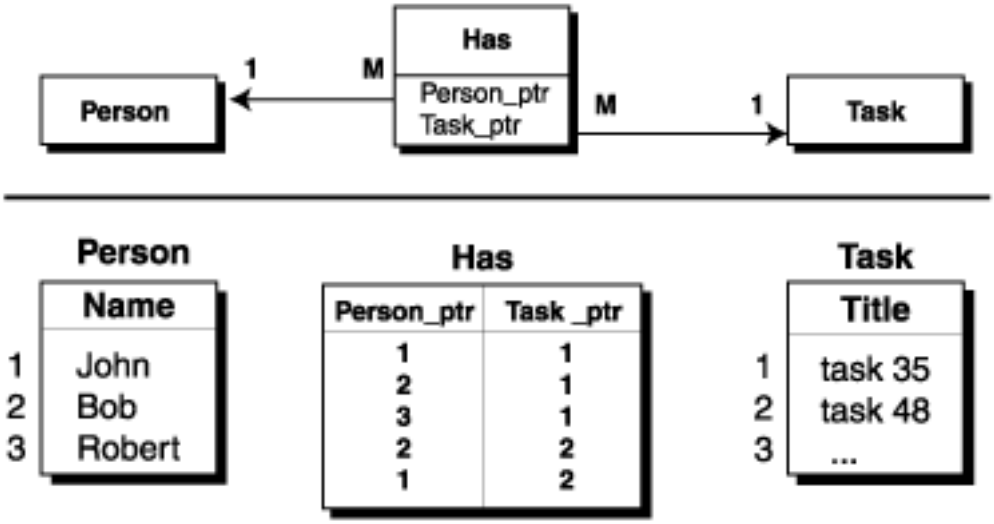
Example 4 (3 tables, Many to Many relation)

In this example, we demonstrate how to use the ObjectPtr field to establish a Many to Many relation between 2 Tables.

This example is based on the project of example 3.

In a real life situation, several Persons will probably work on the same task. For example, Bob, John and Robert can work on “task 35”, and at the same time Bob and John can work on “task 48”.

The Database of example 3 can not handle such a situation. The database Design is shown in the following picture:



As you can see, to establish a Many to Many relation between the Person and Task tables, we need to create an additional Table with 2 fields - which plays the role of pointers. Such a database structure allows us to connect one Person to many Tasks, and one Task to many Persons.

1) Since we need a third table, we should create in the our project another class linkHas as a child of VBaseObject. We assign to it the prefix “link” to emphasize that this table does not keep user’s data: it is used only to build a relation (link) between 2 tables.

2) Add to MyDataBase class a reference to this Table (BaseObject):
   mLinkHas as linkHas

3) Add to linkHas class properties for fields of the Table:
   PersonPtr as VObjectPtr
   TaskPtr as VObjectPtr

4) Add to linkHas class a constructor to make objects of the fields:

```
LinkHas.linkHas:
Sub linkHas(inDataBase as MyDataBase)
        // Initialise class members
        // Note, this is not required, but it is often handy to have a reference from
        // a table to its database
        mDataBase = inDataBase

        // Set name of the Table
        name = "has”

        // Make fields:
        // Note, BaseObjects mPerson and mTask have already been created in the
        // the constructor of MyDataBase. This is very important!
         PersonPtr = new VObjectPtr( "person_ptr”, inDataBase.mPerson, kV_Cascade )
        TaskPtr = new VObjectPtr( "task_ptr”, inDataBase.mTask, kV_Cascade )
End Sub
```

Note that in this constructor, we need to access the 2 Tables of our database – Person and Task. That is why it is better to pass an object of the database via a parameter.

5) Since we have made a copy of the project of Example 3, now we can remove the field PersonPtr from the class boTask. Delete it as a property and remove it from the constructor of boTask.

6) Change slightly the method AddTestRecords of class boTask to fill 3 tables with data:

```
boTask.TestAddRecords:
Sub TestAddRecords( PersonID as integer )
        dim i as integer

                // In the following loop we add 4 Task records for each Person.
                for i = 1 to 4
                        // First we add a record to the Task table
                        SetBlank

                        mName.value = "task " + str(i)

                        mStartTime.Set( i, 0, 0 )
                        mFinishTime.Set( i + 6, 20, 40 )

                        mStatus.value = i
                        AddRecord

                        // Now we add a link-record between Person and Task
                        mDataBase.mLinkHas.PersonPtr.value = PersonID
                        mDataBase.mLinkHas.TaskPtr.value = GetRecID
                        mDataBase.mLinkHas.AddRecord
                next
End Sub
```

Working with Example 4

1) Run the project of example 3

2) Make a new database on disk using New from the File menu.

3) Choose “Add 900’000 records” from the Test menu.

4) Go to the Query window.
Now you can try a SQL query on both tables:

```
SELECT *
FROM Person, Has, Task
WHERE  byte_fld < 10 AND status = 1 AND
            person.id = has.person_ptr AND has.task_ptr = task.id
ORDER BY string_fld DESC
```

Again, with Valentina you can skip the specification of a LINK condition:

```
SELECT *
FROM Person, Has, Task
WHERE byte_fld < 10 AND status = 1
ORDER BY string_fld DESC
```

Example 5 (dynamic database structure)

In this example, we show quite a different way of using Valentina for REALbasic. It is based on the project of example 4.

We do not recommend using this method normally. However, it is useful if you really want to develop a DBMS-like application, i.e. an application which can open and work with a database with an unknown structure.

We will not define the structure of the database via classes. We also will not create classes for the Tables or create properties for the fields.

We will use only one class MyDataBase. Actually we could even skip this class entirely and instead put all methods in a separate module.

So how will we create a database? For this we create several methods which work together. We will use the CreateBaseObject() and CreateField() methods.

```
Sub CreateStructure()
	dim obj as VBaseObject

	obj = CreateBaseObject( "Person” )
	obj = CreateBaseObject( "Task” )
	obj = CreateBaseObject( "Has” )

	CreateFieldsOfPerson
	CreateFieldsOfTask
	CreateFieldsOfHas
End Sub

Sub CreateFieldsOfPerson()
	 dim fld as VField

	dim thePerson as VBaseObject
	thePerson = BaseObject( "person” )

	fld = thePerson.CreateField( "bool_fld”,	kV_TypeBoolean )
	fld = thePerson.CreateField( "byte_fld”,	kV_TypeByte )
	fld.Nullable = true

	fld = thePerson.CreateField( "short_fld”,	kV_TypeShort )
	fld = thePerson.CreateField( "ushort_fld”,	kV_TypeUShort )
	fld = thePerson.CreateField( "long_fld”,	kV_TypeLong )
	fld = thePerson.CreateField( "ulong_fld”,	kV_TypeULong )
	fld = thePerson.CreateField( "float_fld”,	kV_TypeFloat )
	fld = thePerson.CreateField( "double_fld”,	kV_TypeDouble )
	fld = thePerson.CreateField( "string_fld”,	kV_TypeString, 30 )
	fld = thePerson.CreateField( "date_fld”,	kV_TypeDate )
	fld = thePerson.CreateField( "time_fld”,	kV_TypeTime )
End Sub
```

```
Sub CreateFieldsOfTask()
        dim fld as VField

        dim theTask as VBaseObject
        theTask = BaseObject( "task” )

        fld = theTask.CreateField( "name”, kV_TypeString, 20 )
        fld = theTask.CreateField( "start_time”,   kV_TypeTime )
        fld = theTask.CreateField( "finish_time”, kV_TypeTime )
        fld = theTask.CreateField( "status”, kV_TypeByte )
End Sub


Sub CreateFieldsOfHas()
        dim fld as VField

        dim theHas as VBaseObject
        theHas = BaseObject( "has” )

        fld = theHas.CreateField( "person_ptr”, kV_TypeObjectPtr,
                                        BaseObject("person”),  kV_Cascade )
        fld = theHas.CreateField( "task_ptr”, kV_TypeObjectPtr,
                                        BaseObject("task”), kV_Cascade )
End Sub
```

The main difference of working with a dynamic database is that your methods get references based on class VField, but not on VByte or VString. This means you cannot use the property Value of that class to read / write the value of the fields. For this purpose you can only use methods of the class Vfield: SetString() and GetString().

Taking this into account, we should change all methods which use the property Value of the fields. Besides, since we do not have static references to the fields of the BaseObjects, we need to obtain them dynamically each time using method VBaseObject.Field() (see next page for an example).

If you will need to get access to properties of a specific class such as VString, then you need to use type casting:

```
        dim fld as VField
        dim StrFld as VString

        fld = BaseObject(“person”).Field(“string_fld”)
        if( fld.type = kV_TypeString ) then
                StrFld = VString( fld )
        end if
```

```
Sub AddRecordsToPerson(N as integer)
	dim i as integer
	dim thePerson as VBaseObject

	thePerson = BaseObject( "person” )

	for i = 1 to N
		thePerson.SetBlank

		if( (i Mod 2) = 1 ) then
			thePerson.Field("bool_fld”).SetString( "1” )
		else
			thePerson.Field("bool_fld”).SetString( "0” )
		end if

		if( i Mod 23 <> 0 ) then
			thePerson.Field("byte_fld”).SetString(    Str(i Mod 256) )
		else
			// Do nothing, so field will have NULL value.
		end if

		thePerson.Field("short_fld”).SetString( Str(- i))
		thePerson.Field("ushort_fld”).SetString( Str(i))
		thePerson.Field("long_fld”).SetString( Str( -2000000 * i))
		thePerson.Field("ulong_fld”).SetString( Str(2000000 * i))
		thePerson.Field("float_fld”).SetString( Str(3.1415 * i))
		thePerson.Field("double_fld”).SetString( Str( 3.1415 * i * 10000))
		thePerson.Field("string_fld”).SetString( “line “ + Str(i) )

		d.Year = (i mod 2500) + 1
		d.Month = (i mod 12) + 1
		d.Day = (i mod 31) + 1
		thePerson.Field("date_fld”).SetString( d.shortDate )

		d.Hour = i mod 12
		d.Minute = i mod 60
		d.Second = i mod 60
		thePerson.Field("time_fld”).SetString( d.ShortTime )

		thePerson.AddRecord
	next

	 thePerson.Flush
End Sub
```

We do not recommend using this technique in general because it is significantly slower than using classes. There are 2 reasons for that:

1) There is great overhead in converting field context to / from strings.

2) We must spend time obtaining references to fields via the methods of Field(). Using classes, we have direct references to fields stored as properties.

As you can test for yourself, example 5 works about 20 times (!!!) slower than example 4 on the task of adding records. Time for searching and sorting is the same, because this is work done in the Valentina database engine.

Example 6 (optimized)

This example is very similar to Example 5, except that the code is optimized.

The only difference is that we have changed the following methods:

BaseObject(Name as String)
Field(Name as String)

To these methods, everywhere where this is possible:

BaseObject(Index as integer)
Field(Index as Integer)

As we have noted before, using these methods adds some overhead. Accessing BaseObjects and Fields by index is faster than by name, because we avoid performing string comparison.

To be able to do this we add to our project the module MyConstants. In this module we add constants for each BaseObject and Field:

```
kPerson   = 1
kTask     = 2
kHas      = 3
....
```

So we change the following code:

BaseObject(“person”)

to this:

BaseObject(kPerson)