Introduction to Relational Tutorials

The tutorials that follow are designed to be short, simple and quick to learn. They are written for first time users, as well as those who need to refresh their understanding of how specific features work. In general, each tutorial builds on the knowledge you gained from the previous one. Each concentrates only on the material being introduced, and includes suggestions and tips on how to integrate Jovis into your applications. With a few exceptions, the amount of time required for each tutorial should be only a few minutes.

Each tutorial concludes with a summary of what should have been learned. If you find that you didn't fully understand the tutorial, you should go back and review the information until it is completely learned. We have tried not to repeat material, or re-explain concepts presented previously. Therefore, you must be secure in having learned everything as it is summarized at the end of each tutorial.

Throughout the manual we use the term "Shell Application" to refer to HyperCard, SuperCard, Director, and Oracle Media Objects. It simply means whichever scripting application you're using that has an XCmd interface. It is this interface that links Jovis with your scripts. In general, the tutorials spend very little time discussing the parameters for each command. (You will find that the syntax reference section of the manual is devoted to explaining each parameter in detail.) Instead, the emphasis is on the order in which the commands should be called, and special circumstances that you need to be aware of.

We deliberately avoided providing these tutorials on disk. As everyone knows, learning is best done by doing, and while some scripting is required, it is not so much that it distracts from what is being explained. Learning and using Jovis is a great deal of fun, and we think you will quickly catch on to the ideas and information presented throughout the tutorials. In fact, once you've caught onto using Jovis for managing information, we think you'll wonder how you ever got along without it.

All of the tutorials use the same database information, which is for a retail store whose customers and their purchases are being maintained. Basically, the database holds the customer's name, address, account ID, and the purchases made. The tutorials begin by creating the database file along with its format. Next, creating, storing and retrieving customer information is introduced, followed by how to create selections, or subsets, from the entire database. All of the customer's information is provided in text file format, and can be imported using the 'ImportData' command. (The "Importing and Exporting" tutorial will step you through this phase.)

If you are new to Jovis and need to get started quickly, the core, or foundation tutorials are 1-8. With these, you can get by, but you will eventually want to explore the rest of Jovis. There are many features and capabilities built into Jovis. At first, they may seem superfluous, but as your applications develop greater complexity, you'll discover that Jovis is well designed to meet each level of your application's requirements.

Storing and retrieving multimedia objects, such as sounds, pictures, and Blobs (Binary Large OBjects), is done using the Architectural commands. You should go through the first two tutorials concerning the Architectural level, before you start working with multimedia objects. A tutorial using Relational and Architectural commands together is provided as a final tutorial.

Tutorial 1

Creating, Opening and Closing A Data File

Definitions

Shell Application: We use the term "Shell Application" to refer to HyperCard, SuperCard, Director and Oracle Media Objects. It means whichever programming application you are using that has an XCmd interface.

Global File Identifier: A shell application global used as an identifier telling Jovis which data file you want to work with, as well as where in memory the relevant information is located. (We will use "myDB" throughout the tutorials. When you move beyond the tutorials, you can, of course, call your global file identifier anything you want.)

Function handler: A script used to calculate and return a value.

Initialization: Initialization means that the 'CreateCollection' and 'OpenCollection" commands "call back" to the shell application, Jovis knows which data file is being referred to. What happens is that you are passing the name of a global to Jovis. Jovis then examines the global to make sure it contains the word "Jovis", and then sets up the link to the data file.

Before You Begin

This tutorial explains how to correctly create, open, and close a Jovis database file from your scripts. We assume that you know how to use a Macintosh computer, especially naming and locating files on your hard drive. You must also have a basic comprehension of scripting techniques. Such as creating buttons and fields, as well as writing scripts using "if", "else", "repeat with", etc.

It is important for you to begin scripting the commands as they are introduced. First, launch your shell application, and create a new project or stack, which will be your tutorial project or stack. If you haven't already done so, use the Jovis Installer to install Jovis into that project or stack. This will allow you to start using Jovis with that project or stack. (See the Installation documentation for further details.)

A simple test to see if Jovis is installed is this one-line script:

```
put Jovis("%")
```

If Jovis is not installed, your shell application will return a script error, such as "Can't understand "Jovis", or something similar. If Jovis is installed, you will get the current version:

```
Jovis™ Version,1.0.3
```

You are now ready to write the tutorial scripts to that stack or project. Type each one in as it is introduced, and before long you will have a functional demo.

Overview

This first tutorial introduces five commands: CreateCollection, OpenCollection, CloseCollection, Startup, and Shutdown. The concept and use of a shell application global as a file identifier is introduced. The special handling of errors and warnings for these commands is also explained. We then explain the Startup and Shutdown commands and how they are used.

Getting Started

CreateCollection/OpenCollection

The first script of this tutorial will be used to create the "DASCO Database" file. Jovis uses the term "Collection" to mean a data file or database. They mean the exactly the same thing.

The first step in creating or opening a collection is to Initialize a shell application global which will serve as a file identifier. The global file identifier tells Jovis which data file you want to work with.. For these tutorials, the global file identifier will be 'myDB'; the data file to which it refers will be called "DASCO Database."

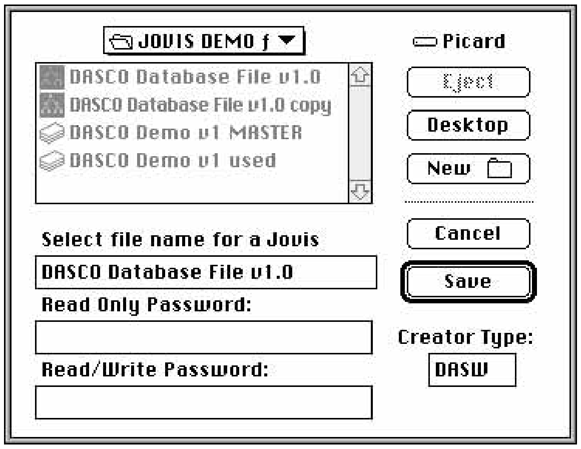
We will use a function script to verify the creation of the data file. This returns "false" if the data file is not successfully created, and "true" if everything works as intended. Using a function script provides a clean way to call scripts that indicate the success or failure of creating a data file. For example:

```
on mouseUp
   if CreateNewFile() = “false” then go home
end mouseUp
```

Here is the script for initializing a global and calling the ‘CreateCollection’ command. You should type this into your tutorial stack.

```
function CreateNewFile
   global myDB, JovisErrorCode
   —
   if myDB = empty then
      —
      put “Jovis” into myDB
      —
      get Jovis(“CreateCollection”,”myDB”)
      if item 1 of JovisErrorCode = “error” then
            answer JovisErrorCode
            put empty into myDB
            return “false”
      else if item 1 of JovisErrorCode = “Warning” then
            answer JovisErrorCode
            put empty into myDB
            return “false”
      end if
      —
   else
      return “false”
   end if
   —
   return “true”
   —
end CreateNewFile
```

This is a fairly straightforward script. When it is executed, it brings up the Macintosh standard "Put File" dialog (See below.), by way of the ‘CreateCollection’ command, to let you enter a name and set the location for a new data file.



Select file name for a Jovis
DASCO Database File v1.0
Read Only Password:
Read/Write Password:
Creator Type:
DASW

Execute the script, enter the name “DASCO Database File v1.0”, and set a location on your hard drive.

The script has several important points that you need to be aware of:

1. Line 2 has declared two globals: ‘myDB’ and ‘JovisErrorCode’. In this example, ‘myDB’ is going to be used as the global file identifier. This identifier tells Jovis which data file you want to work with, as well as where in memory the relevant information is located. (We will use ‘myDB’ throughout the tutorials. When you move beyond the tutorials, you can, of course, call your global file identifier anything you want.)

   Remember to declare the global file identifier AND use the ‘JovisErrorCode’ global whenever you create or open a data file using the ‘CreateCollection’ and ‘OpenCollection’ commands. Tutorial two will go into greater detail about error handling, but it is important to include the ‘JovisErrorCode’ global for this function script.

2. Line 4 of our ‘CreateNewFile’ example tests whether ‘myDB’ is empty. This is important; if this global is already in use (not empty), the next line in our script will re-initialize the global, and we will lose the original identifier. Once we are sure that ‘myDB’ is empty, we can initialize it.

3. In Line 6, we initialize ‘myDB’ by putting the name “Jovis” into it. This initialization means that when the ‘CreateCollection’ and ‘OpenCollection” commands “call back” to the shell application, Jovis can figure out which data file is being referred to. (Jovis examines the content of the global to make sure it equals the word “Jovis”, and then sets up the link to the data file.)

   NOTE: If you inadvertently change what this global contains, you lose the ability to access the data file no matter what command you try to use, including the ‘CloseCollection’ command! If this happens, your only alternative is to call the ‘ShutDown’ command and quit the shell application, relaunch it and then re-open the database. **Treat your global file identifiers with special care and everything will be fine**. Creating or opening a second data file is simply a matter of using a new and different global name. The only limitation is how much memory you have, and the size of your hard drive. Once the file is created or opened, you do not have to declare the global in your scripts.

4. Line 8 is our ‘CreateCollection’ call. Notice that the global identifier is in quotes. This means that we are passing the *name* of the global to Jovis, NOT the *contents* of the global. Every call to Jovis requires that the global be in quotes.
   Once the ‘CreateCollection’ command has the name of the global, it will handle the details of setting up the identifier for future use by any of the other commands. All you have to do is remember to pass the name of the global in quotes.

5. Lines 9 through 12 demonstrate how to handle errors that could occur. Since the 'CreateCollection' command has been called, we need to check for errors that may have occurred. This is done by checking the first item of the global 'JovisErrorCode'. If there is an error, item 1 of the 'JovisErrorCode' global will equal "error". The Answer dialog box will display the error message contained in the global 'JovisErrorCode' (line 10). In line 11, 'myDB' is emptied, and we exit from the script by returning "false". If there is no error, item 1 of the 'JovisErrorCode' global will equal "no error".

   Similarly, in lines 13 through 16 we check for warnings. If item 1 of the 'JovisErrorCode' global equals "warning", the Answer dialog box displays the warning message, "myDB" is emptied, and we exit the script by returning "false". A warning message is issued when the user clicks the cancel button from the standard "File Put" dialog'; this is the most frequent warning message. If there is no warning, item 1 of the 'JovisErrorCode' global will equal "no error".

   The various error and warning messages that may occur are listed later in this Tutorial.

This completes everything concerning our 'CreateNewFile' example. The 'OpenCollection' command is done similarly. That script is provided in the "Script Examples" section of this Tutorial; you should type it into your stack as well.

Both the 'CreateCollection' and 'OpenCollection' commands have several other parameters, which are all optional. These are explained in the Syntax Reference section of the Jovis manual.

CloseCollection

The correct way to close a data file requires the use of the 'CloseCollection' command. Here is our example for you to type in:

```
on CloseDBFile
   global myDB,JovisErrorCode
   if myDB is not empty then
      if myDB is not “Jovis” then
         —
         get Jovis(“CloseCollection”,”myDB”)
         if item 1 of JovisErrorCode = “error” then
         answer JovisErrorCode
         exit CloseDBFile
         end if
         —
         if myDB = “Jovis” then
            put empty into myDB
         end if
         —
      end if
   end if
end CloseDBFile
```

This will close your data file without shutting down Jovis.

In line 2, we have once again declared our global file identifier “myDB” just as in the ‘CreateCollection’ example above.

Lines 3 and 4 check to make sure that the identifier is not empty, and that it has not already been closed. When you call the ‘CloseCollection’ command (line 6), Jovis will save any last minute changes and then close the file. It then resets the global identifier to the name “Jovis”.

In line 12, we test that no errors occurred by making sure that the identifier now equals “Jovis”. If so, line 13 puts empty into the global. This prepares it for reuse if necessary. Once a datafile has been closed, it cannot be accessed by any of the Jovis commands, until, of course, it is reopened.

NOTE: You MUST “shut down” Jovis before closing your stack, or quitting your shell application. See the ‘ShutDown’ command below for further instructions.

If any serious errors occur, our error and warning handlers will have caught them. (The next tutorial will explain these handlers.)

Using the ‘StartUp’ and ‘ShutDown’ commands

These commands are used by the Jovis system itself. The ‘StartUp’ command is optional, and is meant for use with large databases. The ‘ShutDown’ file is very important and should be called when you are about to quit your application.

StartUp

The optional ‘StartUp’ command is used when your data files have several indexes and/or the file indexes are quite large. For example, data files with over one hundred thousand records, and/or more than four or five indexes should use this command. It allocates a larger cache than the default, which is 512k. Low memory warnings and out-of-memory errors often indicate that the cache is too small.

To allocate a larger cache, you must call this command once before any ‘OpenCollection’ or ‘CreateCollection’ commands are made. It automatically creates and initializes a shell application global called ‘JovisPrivateData’. NEVER alter the contents of this global; it is strictly for use by Jovis. If this command is not used, the ‘OpenCollection’ or ‘CreateCollection’ commands will automatically setup the ‘JovisPrivateData’ global. Here’s a script that uses the ‘StartUp’ command:

```
on SetJovisStartUp
   global JovisErrorCode
   put 512000 * 2 into CacheSize  —> 1meg cache
   get Jovis(“StartUp”, CacheSize)
   if item 1 of JovisErrorCode = “error” then
      answer JovisErrorCode
   end if
end SetJovisStartUp
```

ShutDown

The ‘ShutDown’ command is required. It shuts down the Jovis system before you quit your shell application. This consists of a simple script, such as:

```
on ShutDownJovis
   global JovisPrivateData
   if JovisPrivateData ≠ empty then
      get Jovis(“ShutDown”)
   end if
end ShutDownJovis
```

Notice that we check that the Jovis system has not been shut down already. We do this by verifying that the ‘JovisPrivateData’ global is not empty. If the command is successful, the global is cleared, and is ready to be used again if necessary.

Script Examples

Here is a ‘OpenCollection’ script that could be used for general purposes:

```
on DoOpenFile
   global myDB,JovisErrorCode
   if myDB = empty then
      — initialize by putting the name “Jovis” into a global
      put “Jovis” into myDB
      get Jovis(“OpenCollection”,”myDB”)
      if item 1 of JovisErrorCode = “error”  then
         answer JovisErrorCode
         put empty into myDB
         exit DoOpenFile
      end if
      if item 1 of JovisErrorCode = “warning” then
         answer JovisErrorCode
         put empty into myDB
         exit DoOpenFile
      end if
   else
      beep
```

```
         answer “Error: Invalid global identifier.”
      end if
   end DoOpenFile
```

Errors and Warnings that may appear in the ‘JovisErrorCode’ global

Script Errors:
Can’t recognize or understand “Jovis” —> Jovis is not installed. See the installation documentation of details.
Expected “)” but found “,” —> missing or unbalanced quotes

General Errors:
Not a valid global name.
Not enough parameters.
Not a Jovis relational file.
Required resources missing.
Not enough memory.

Network Errors:
Server not responding.
File not found. Access to the file has been denied from the server.
File in use. File has been opened in exclusive mode.
File already open. Tried to create a file with same name as one already open at the server.

Warning Messages
Cancel selected by user.
File is open read-only.
Memory Low No Reserve. Save and quit application.

What You Should Have Learned

You should now have a clear understanding of how to create, open and close a Jovis database file using the ‘CreateCollection’ ,’OpenCollection’, and ‘CloseCollection’ commands. This should include the concept and importance of using the global file identifier.

You should also understand that you must use the JovisErrorCode’ global for the ‘CreateCollection’ and ‘OpenCollection’ commands. The provided scripts give examples of how to handle errors which might occur with these commands.

Also covered were the two commands, ‘StartUp’ and ‘ShutDown’, which control the Jovis caching system. While the ‘StartUp’ command is optional, and generally used for large data files, the ‘ShutDown’ command is a required command. ‘ShutDown’ must be called when it’s time to quit your application.

What to learn next

We have already discussed using the 'JovisErrorCode' global for the 'CreateCollection' and 'OpenCollection' commands. Jovis contains a very effective error and warning system which is used for all of the remaining Jovis commands. The next tutorial will address how to use this system and why it is so important.

Tutorial 2

Error and Warning Handlers

Before You Begin

This tutorial explains how Jovis keeps you informed of any errors and warnings that may occur. Error handling is essential for debugging and proper monitoring of program execution. It is a requirement, from the day you begin writing an application, until your end-users finish using it.

Fortunately, Jovis provides a very effective system for reporting errors immediately, with the information needed to solve the problem quickly and easily. Give yourself enough time to fully learn this tutorial. It will save you time in the end, as well as preventing file corruption, and possible system restarts.

Overview

The error and warning system contained in Jovis immediately provides you with information about errors or warnings when they occur. Both errors and warnings provide information about what command is reporting the error, the name of the data file, what the error is, and an error or warning ID code number. In addition, you can log errors and warnings to a text file at any time, as well as abort your scripts if necessary.

Jovis provides an override mechanism so that you can perform error checking locally in your scripts without the end-user becoming involved. You would use this technique when an error might be generated, but you want the end-user to see a different message. This override capability will introduce the 'SetProperty' command.

You should immediately begin using the error and warning system that is presented here. All of the commands in Jovis use this mechanism, except the 'CreateCollection', 'OpenCollection', 'DBOpen' and 'DBInit' commands, which use the 'JovisErrorCode' global. By implementing and trying out the scripts in each phase of this tutorial, you can be sure that you have learned this important feature.

Getting Started

Jovis primarily reports errors and/or warnings by sending a message to your script, starting at the 'Card' level. The error message that is sent is "JovisErrorMsg", and the warning message sent is "JovisWarningMsg". Both of these messages MUST be present in your scripts as function handlers. Here is the sample error function handler, which you should now put into the highest message hierarchy, i.e. Stack script or Project script, of your tutorial stack or project:

```
function JovisErrorMsg theFile, theRel, theCmd, theMsg, ErrCode
   beep
   put "ERROR!" into temp
   put theMsg into line 3 of temp
   put "Command:"&&theCmd into line 5 of temp
   if theFile is empty then put "Unknown at this time." into theFile
   put "File:"&&theFile into line 6 of temp
   if theRel is not empty then put "Relation:"&&theRel into line 7 of temp
   if ErrCode is not empty then
      put "Code Number:"&&ErrCode into line 8 of temp
   end if
   answer temp with "Exit Script"
   return "Stop"
end JovisErrorMsg
```

Note these important points:

1. To display the information being provided by the incoming parameters, we format a local variable called 'temp'. Once 'temp' has been "filled in", we call the answer dialog to display the content of the 'temp' variable.

2. When the user clicks the answer dialog button called "Exit Script", the function handler returns the word "Stop" to Jovis. When this occurs, Jovis immediately causes the shell application to abort further execution of your scripts. At this point you must investigate what caused the error and fix it immediately.

3. The five parameters includedwith these messages indicate:

theFile:	the name of the file on the hard drive which contains the error;
theRel:	the name of the relation, we will discuss this term in a future tutorial;
theCmd:	the name of the command which triggered the error;
theMsg:	a description of the error, if available; and
ErrCode:	the ID code for the error, which is listed in the back of the Jovis manual.

Here's an example of the warning function handler:

```
function JovisWarningMsg theFile, theRel, theCmd, theMsg, ErrCode
   put "WARNING!" into temp
   put theMsg into line 3 of temp
   put "Command:"&&theCmd into line 5 of temp
   if theFile is empty then put "Unknown at this time." into theFile
   put "File:"&&theFile into line 6 of temp
   if theRel is not empty then put "Relation:"&&theRel into line 7 of temp
   if ErrCode is not empty then
      put "Code Number:"&&ErrCode into line 8 of temp
   end if
   answer temp with "Exit Script","Continue"
   if it = "Exit Script" then return "Stop"
end JovisWarningMsg
```

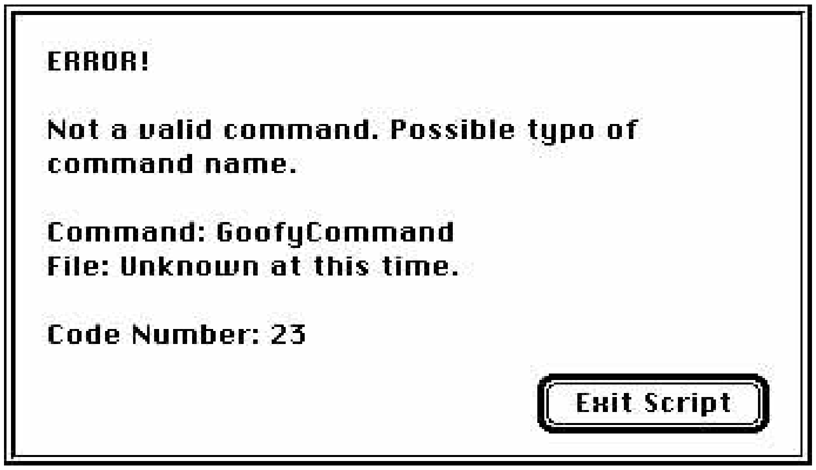
Again, we simply format a local variable called 'temp' and display its contents using the answer command. With the Warning message, however, the answer dialog uses two buttons: one is "Exit Script", the other is called "Continue", which is the default. Because warnings are not considered serious enough to require you to abort your scripts, the user has the option of allowing their scripts to continue execution. However, they may abort by clicking the "Exit Script" button.

Let's take some time now to implement and test these two function handlers. First, open your tutorial project and type both handlers into the highest message level possible, i.e. the Stack script or Project script.

Now, we will purposely generate a Jovis error. Using the message box, type the following and press the return key:

```
get Jovis("GoofyCommand")
```

If nothing at all happened, you have mistyped the error handler script, and should double check it against our example. If the script is correct, you will see this answer dialog:



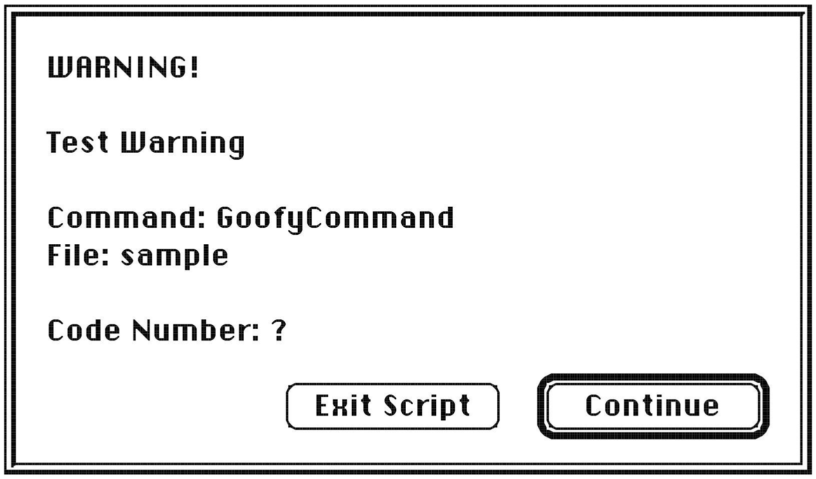
```
ERROR!

Not a valid command. Possible typo of
command name.

Command: GoofyCommand
File: Unknown at this time.

Code Number: 23
```

Press the return key, and move on to the next step.

Now we will test that the warning function handler to see if it is working correctly. There is no simple way to cause Jovis to send a warning message, as we did with the error message, so we will call the handler directly using the message box. Type in the following and press the return key:

```
get JovisWarningMsg("sample","","GoofyCommand","Test Warning","?")
```

This answer dialog should appear:



```
WARNING!

Test Warning

Command: GoofyCommand
File: sample

Code Number: ?
```

Press the return key to remove the answer dialog from your screen.

Overriding the Error and Warning Messages

It is possible to override either or both of the error and warning function handlers. The 'SetProperty' command has two properties, one called "ErrorMsg", the other "WarningMsg". By setting either of these properties to a new function handler name, you can override the standard function handlers, and call your own handlers. For example:

```
on mouseUp
   get Jovis("SetProperty","myDB","ErrorMsg","InvalidCriteria") -- replace
   put "Field Last_Name = [Jones]" into aCriteria
   get Jovis("CountMatches","myDB","Customers",aCriteria,"true")
   get Jovis("SetProperty","myDB","ErrorMsg","JovisErrorMsg") -- restore
end mouseUp
```

```
   function InvalidCriteria theFile, theRel, theCmd, theMsg, ErrCode
      answer "Entered criteria is not valid."
   end InvalidCriteria
```

In line 2, we set the error message handler to a new handler called "InvalidCriteria", which is shown above. In line 4, we called the 'CountMatches' command, whose description can be found in the syntax reference section of this manual.. (We will discuss criteria in a later tutorial) If the criteria is invalid, Jovis will send the "InvalidCriteria" message. The new function handler will respond to it by displaying the answer dialog. Line 5 then resets the error message back to our standard handler.

Another way to override the error and warning handlers is to set the message to empty, then use the 'JovisErrorCode' global discussed in Tutorial 1. If item 1 of this global equals "error", you can respond as appropriate. The message is reset back to the standard handler before exiting the script. Here is an example: of using the 'JovisErrorCode' global instead of a handler:

```
on CountRecords
   global JovisErrorCode
   get Jovis("SetProperty","myDB","ErrorMsg","") -- clear function handler
   put "Field Last_Name = [Jones]" into Criteria
   get Jovis("CountMatches","myDB","Customers",Criteria,"true")
   if item 1 of JovisErrorCode = "error" then
      answer "Error while trying to count records."
   end if
   get Jovis("SetProperty","myDB","ErrorMsg","JovisErrorMsg") -- restore
end CountRecords
```

IMPORTANT: Absolutely never override an error or warning message and not check for errors. When you override, always use either an alternative function handler or the 'JovisErrorCode' global. If you do not, the results are unpredictable, and could lead to serious problems, including data file corruption and system crashes.

Script Examples

This 'LogError' handler is a straight forward way to write errors that occur to a text file:

```
   on LogError aMsg
      global errLogPathName
      if errLogPathName is empty then
         ask file "Please enter error log name and set it's location."
         if the result is "cancel" then exit logError
         put it into errLogPathName
      end if
      open file errLogPathName
      write the abbrev date&" - "&the long time&" - "&aMsg&return
            to file errLogPathName at EOF
      close file errLogPathName
   end LogError
```

Related Error Messages

"Function handle doesn't execute"

If either of the error or warning function handlers contains a scripting error, the function will not execute, AND no scripting error will be reported by the shell application! This is a shortcoming of scripts called by external code modules.

If you suspect that an error or warning function handler may contain a scripting error, try calling it from the message box with "fake" parameters. In this way, the shell application will assist you in finding the problem.

What You Should Have Learned

You should now understand how to implement the standard error and warning function handlers for Jovis. The importance of these handlers should be clear to you. Additionally, you should understand how to override either handler with a different message, or disable it and use the 'JovisErrorCode' global.

At this point, you can call any of the Jovis commands knowing that errors and warnings will be reported to you immediately.

What to learn next

Now that you have learned about handling errors and warnings, you are ready to start creating the structure needed for storing and retrieving records. The next tutorial will step you through creating Relations, Fields, and Indexes.

Tutorial 3

Creating Database Structures

Before You Begin

This tutorial explains how to create the framework or database structures in a Jovis data file. You have already created a new data file, the "DASCO Database"; you should also have the scripts to open, close, and shut down your data base. You need to understand and have fully implemented the error and warning function handlers. If you don't have all these scripts ready to use, you should go back and create them now. If you need help, review to the previous tutorial.

Overview

This tutorial will first introduce the commands 'CreateRelation', 'CreateFields', and 'CreateIndexes'.

The database structures we are going to create are for our "DASCO Database". When we are done, we will be ready to begin creating records. While the three "Create" commands are straightforward, you do need to make sure that you are using the error and warning function handlers, so you will catch any errors, such as low memory situations or possible hard drive problems.

Getting Started

CreateRelation

We will begin with the 'CreateRelation' command. In Jovis, a relation is simply the name used to organize a group of fields. It is like a table structure that uses columns and rows to organize and maintain your information. Each row is a record, and each column is a field. A record consists of a series of fields, and each field describes a piece of information. For example, here is a record from the relation with the three fields 'First_Name', 'Last_Name', and 'Customer_ID'.

"John,Smith,14432"

Here's an example of using the 'CreateRelation' command:

```
get Jovis("CreateRelation","myDB","Customers")
```

'CreateRelation' takes only three parameters: the command name; the global file identifier; and the name you have chosen for the new relation.

Limitations

Each relation can contain up to 512 fields, and can hold an unlimited number of records. (Because of the amount of processing done for each field, a more practical limit of about 150 fields is advised. Hardware may limit the number of records, but Jovis will not.) It is important to realize that fields within a record are strictly associated with only one relation. A relation never "shares" or holds combinations of data with other relations. (It is possible to "join" relations together to access different combinations of fields. This will be explained in Tutorial 12).

Assuming that no errors were returned, you are now ready to call the 'CreateField' command.

CreateField

Data Types

The 'CreateField' command requires that you know the type of data you plan on storing. For example, a last name field would use the 'TEXT' data type. Jovis provides four types of data: 'TEXT', 'NUMBER', 'DATE', and 'LOGIC' (often referred to as "true" or "false"). Using the correct data type is important, because when you search by a given field, it must be in the correct sorted order. Numbers are sorted in numeric order, whereas text is sorted by alphabetic order. Fields that contain a mixture of numbers and text, such as an address, use should the 'TEXT' data type. You should also refer to the section in this manual concerning restrictions on names.

Let's continue with our "DASCO Database". We know that our 'Customer' relation will need a first name field. So, we use the 'CreateField' command for creating this field:

```
get Jovis("CreateField","myDB","Customers","First_Name","TEXT")
```

The third parameter is the name of the relation for which we are creating the new field.

The fourth parameter is the name of the field that we are creating. Each field name in a given relation must be unique and consist of ASCII alphanumeric characters; the underscore character may also be used. Spaces, carriage returns, and control characters are NOT permitted

The fifth parameter is the data type, and must be one of the four types mentioned above.

We want our "Customers" relation to have 11 fields:

"First_Name; Last_Name; Is_Home_Address;Address1; Address2; City; State; Zip; Phone; Customer_ID, and Account_Start."

The Zip, Phone, and Customer_ID are Number fields; Account_Start is a Date field; all the others are Text fields. So we could write:

```
      get Jovis("CreateField","myDB","Customers","First_Name","TEXT")
      get Jovis("CreateField","myDB","Customers","Last_Name","TEXT")
      get Jovis("CreateField","myDB","Customers","Is_Home_Address","LOGIC")
      get Jovis("CreateField","myDB","Customers","Address1","TEXT")
      get Jovis("CreateField","myDB","Customers","Address2","TEXT")
      get Jovis("CreateField","myDB","Customers","City","TEXT")
      get Jovis("CreateField","myDB","Customers","State","TEXT")
      get Jovis("CreateField","myDB","Customers","Zip","NUMBER")
      get Jovis("CreateField","myDB","Customers","Phone","NUMBER")
      get Jovis("CreateField","myDB","Customers","Customer_ID","NUMBER")
      get Jovis("CreateField","myDB","Customers","Account_Start","DATE")
```

This above example is very straightforward. However, because everything is "hard coded", it is not the most efficient way to create our "DASCO" project. A somewhat "cleaner" way of scripting the creation of all these fields would be from an itemized list of fields:

```
   on CreateCustomerFields
      put "First_Name,Last_Name,Is_Home_Address,Address1,Address2,City,State,"&
               "Zip,Phone,Customer_ID,Account_Start" into fieldList
      put "text,text,logic,text,text,text,text,number,
               number,number,date" into fieldTypes
      --
      repeat with x = 1 to number of items of fieldList
         get Jovis("CreateField","myDB","Customers",
               item x of fieldList, item x of fieldTypes)
      end repeat
      --
   end CreateCustomerFields
```

This will certainly create all our fields. However, our field list will be very useful, sometimes even necessary, for use with future commands. We will create a function handler that simply returns the list of fields, or the list of field types. To do this, we use the following script. You should include this in your Tutorial stack or project.

```
   function getFieldList listType
      if listType = "Fields" then
         return "First_Name, Last_Name, Is_Home_Address, Address1,"&
            "Address2, City, State,Zip, Phone, Customer_ID, Account_Start"
      else if listType = "Types" then
         return "text,text,logic,text,text,text,text,number,number,number,date"
   end getFieldList
   function CreateJovisFields RelationName,FieldList,TypesList
      repeat with x = 1 to number of items of fieldList
         get Jovis("CreateField","myDB",RelationName, ¬
                     item x of FieldList, item x of TypesList)
      end repeat
   end CreateJovisFields

   on CreateCustomerFields
      get Jovis("CreateFields","Customers", getFieldList("Fields"), ¬
                  getFieldList("Types"))
   end CreateCustomerFields
```

The 'getFieldList' function handler returns our field lists, and the 'CreateJovisFields' function handler does the actual creating of each Jovis field.

The 'CreateCustomerFields' handler is the caller, or driver for the entire procedure. So, you could type, "CreateCustomerFields" from your message box and the fields for the "Customer" relation would be created.

CreateIndex

The last remaining "create" command is 'CreateIndex'. Indexes are used for retrieving records very quickly searching for them in as efficently as possible. In our "Customers" relation, we are going to want to search by Last_Name, Zip, and Customer_ID. For now, we will only index three fields, so we can use this simple script.

```
1:    on CreateCustomerIndexes
2:       get Jovis("CreateIndex","Customers","myDB","Last_Name","8")
3:       get Jovis("CreateIndex","Customers","myDB","Zip")
4:       get Jovis("CreateIndex","Customers","myDB","Customer_ID")
5:    end CreateCustomerIndexes
```

Line 2 is our call to 'CreateIndex' for the "Last_Name" field. The parameters are: the relation, the global file identifier, the field, and the number of characters from that field we want to index. We will use the first 8 characters from each "Last_Name" field for this index.

Lines 3 and 4, the other two 'CreateIndex' calls, are both NUMBER data types; those use a fixed 10 byte width, so we can omit the fifth parameter.

When creating indexes on TEXT fields, it is best to use as few characters as possible but still reasonably distinguish one entry from another. This helps keep the index as small as possible, without generating too many duplicate entries.

For example, imagine we have the following last names:

```
Jackenson
Jackmund
Jacksmith
Jackson
Jacksten
```

If we index only the first 4 characters, the index would consist of "Jack," which would be too general and contain all five entries. If we instead, indexed on the first 5 characters, we would be using "Jacks", which is somewhat better, but would still contain 3 duplicate entries:

```
Jacksmith
Jackson
Jacksten
```

By using the first 8 characters of the "Last_Name" field, as we're planning to do, our index will certainly use enough characters to maintain a sufficient level of uniqueness. If you are unsure, perhaps because you haven't seen enough input data, it is better to err on the high side. For example, 12 characters is certainly better than, say our original of only 4 characters.

Script Examples

Here we show the 'CreateField' scripts above modified to include creating indexes using the field lists. Notice that the 'getFieldList' function handler has now been extended to return the index names and index sizes. We also added a new function handler named "CreateJovisIndexes". Our original driver, 'CreateCustomerFields' is now called 'CreateCustomerRelation", and now also includes a call to the new function handler "CreateJovisIndexes" as well as the original 'CreateRelation' command call.

Now everything for setting up the "Customers" relation is taken care of. First, we create the relation, then the fields, and immediately following that, we create the indexes.

```
function getFieldList listType
   if listType = "Fields" then --> MUST be same item count as "Types".
      return "First_Name,Last_Name,Is_Home_Address,Address1,Address2,City,"&
               "State,Zip,Phone,Customer_ID,Account_Start"
   else if listType = "Types" then --> MUST be same item count as "Fields" above.
      return "text,text,logic,text,text,text,text,number,number,number,date"
   else if listType = "Indexes" then
      return "Last_Name,Zip,Customer_ID" --> MUST be same item count as "Sizes".
   else if listType = "Sizes" then
      return "12,10,10" --> MUST be same item count as "Indexes" above.
   end if
end getFieldList
```

```
function CreateJovisIndexes RelationName,IndexList,Sizes
   repeat with x = 1 to number of items of IndexList
      get Jovis("CreateIndex","myDB",RelationName,
                  item x of IndexList, item x of Sizes)
   end repeat
end CreateJovisIndexes

function CreateJovisFields RelationName,FieldList,TypesList
   repeat with x = 1 to number of items of fieldList
      get Jovis("CreateField","myDB",RelationName,
                  item x of FieldList, item x of TypesList)
   end repeat
end CreateJovisFields

on CreateCustomerRelation
   get Jovis("CreateRelation","myDB","Customers")
   get CreateJovisFields("Customers", getFieldList("Fields"),
                  getFieldList("Types"))
   get CreateJovisIndexes("Customers", getFieldList("IndexList"),
                  getFieldList("Sizes"))
end CreateCustomerRelation
```

Error Messages That You May Encounter

Not a valid relation name
Relation name too long
Invalid Param Info
Not Enough params
Not Valid Global
Not Enough Memory
Field already exists
Not a valid field type
Field name too long
Field does not exist in relation
Not a valid field name
Invalid field name chars
Index already exists

What You Should Have Learned

You have now learned all of the basics for creating a relation and its structures. We covered the use of the data types 'Text', 'Number', 'Date', and 'Logic', when calling the 'CreateField' command. Next, we discussed creating indexes, and, how to determine how many characters should be used for maintain a sufficient level of uniqueness for a 'Text' data type. Finally, we provided a complete script example for creating the "Customers" relation for our "DASCO Database".

What to learn next

Now that we have created our "Customers" relation and its structures, it is time to start creating records. You will begin by learning how to load a record's fields with input data and save it to the database.

Tutorial 4

Creating Records

Before You Begin

This tutorial explains how to create records for a given relation. You should already have a "Customers" relation in the "DASCO Database". All of the fields and indexes for this relation should have been created. The Jovis error and warning function handlers should already be implemented. The concepts of a shell application global file identifier, and what a relation is, as well as how to create field and indexes should be familiar.

Overview

In this tutorial, we are going to show you how to create records, load data into the fields of a record, and then save the record to the datafile. We will also show you how to turn on and off the "AutoSave" feature. If you are using the client/server version of Jovis, you should study the "Transactions" tutorial for information on creating records with "Transactions".

Getting Started

CreateRecord, SetRecordField, and UpdateRecord Commands:

Creating a new Jovis record requires 3 simple steps:

1. call 'CreateRecord' to return a new empty record;

2. use 'SetRecordField' or 'FillRecFields' to load your data into the fields of that record; and

3. call 'UpdateRecord' to save the record to the database.

Here's a quick example of how all this works;

```
1: on CreateNewRecord
2:    put Jovis("CreateRecord","myDB","Customers") into aRecord
3:    put Jovis("SetRecordField","myDB","Customers",
4:                   aRecord,"Last_Name","Smith") into aRecord
5:    get Jovis("UpdateRecord","myDB","Customers",aRecord)
6: end CreateNewRecord
```

Line 2 uses the 'CreateRecord' command to return a new, empty record. In line 3 we then use the 'SetRecordField' command to put data into a given field. In this case, we are putting "Smith" into the "Last_Name" field of the record.

In line 4, we save the record to the data file by calling 'UpdateRecord'. Notice that parameter 4 is our new record which contains our data for the "Last_Name" field .

DO NOT under any circumstances, install data directly into a Jovis record by setting the shell application's 'ItemDelimiter' property. For example:

-- NEVER DO THIS:
```
on DoRecordField aRecor
   set ItemDelimiter to NumToChar(28)
   put "Smith" into item 4 of aRecord
   set ItemDelimiter to comma
end DoRecordField
```

While this might seem harmless, you are more likely to experience incompatibilities when you send the record back to Jovis. Then too, the record format and its delimiters might possibly change in future versions of Jovis. Using 'SetRecordField' is the only way you can be sure of future compatibility.

Let's find a more efficient way to load the record's fields. One way is to have the names of the fields in the "Customers" relation be exactly the same as the fields in the shell application . For example, our "Last_Name" field in the "Customers" relation would also be the same name as a shell application field. We could then reference, say, a background field "Last_Name" as parameter 6 in the 'SetRecordField', like this:

```
   put Jovis("SetRecordField", "myDB", "Customers", aRecord, "Last_Name", ¬
               bkg fld "Last_Name") into aRecord
```

If we take this idea a step further and use our "getFieldList" function handler from the previous tutorial (It's also in the "Script Examples" section below), we could do the following:

```
on CreateNewRecord
   put Jovis("CreateRecord","myDB","Customers") into aRecord
   put getFieldList("Fields") into fldList
   repeat with x = 1 to number of items of fldList
      put Jovis("SetRecordField","myDB","Customers",aRecord,
            item x of fldList,bkgnd fld item x of fldList) into aRecord
   end repeat
   get Jovis("UpdateRecord","myDB","Customers",aRecord)
end CreateNewRecord
```

This is fairly efficient; the repeat loop gets the job done with very little scripting. However, to be even more efficient is the 'FillRecFields' command, which will automatically install the shell application's fields into the new record. Here's our final example using 'FillRecFields':

```
on CreateNewRecord
   put Jovis("CreateRecord","myDB","Customers") into aRecord
   put getFieldList("Fields") into fldList
   put Jovis("FillRecFields", "myDB", "Customers", aRecord, fldList, ¬
               fldList) into aRecord
   get Jovis("UpdateRecord","myDB","Customers",aRecord)
end CreateNewRecord
```

This is the fastest possible solution for installing data into a Jovis record.

AutoSave

The 'UpdateRecord' command incrementally saves the new record to disk as it updates each index in the relation. By incrementally saving your data to your hard drive, Jovis provides the strongest possible security against data loss. However, this 'AutoSave' capability may be too slow in some situations. You can turn "AutoSave" off at any time by setting the fourth parameter of the 'SetProperty' command to "False". While is it turned off, you can control when to actually save your changes to disk using the 'SaveFile' command.

Here's our previous script example now using both the 'SetProperty' and 'SaveFile' commands.

```
on CreateNewRecord
   put Jovis("CreateRecord","myDB","Customers") into aRecord
   put getFieldList("Fields") into fldList
   put Jovis("FillRecFields","myDB","Customers",aRecord,fldList,fldList) ¬
            into aRecord
   get Jovis("SetProperty","myDB","AutoSave","False")
   get Jovis("UpdateRecord","myDB","Customers",aRecord)
   get Jovis("SetProperty","myDB","AutoSave","True")
   get Jovis("SaveFile","myDB")
end CreateNewRecord
```

In line 5, we have turned off the 'AutoSave". Line 6 we 'update' the record in memory, but it is not written to disk. In line 7, we turn the "AutoSave" back on by setting parameter four to "True". (Remember, this command does not actually save any changes. It simply sets an internal flag.) In line 8 we tell Jovis to save all changes made to the data file.
'
You will find that turning the "AutoSave" flag on and off is very useful, particularly during batch processing of records.

The reason for the global file identifier in the 'SaveFile' command is very important for two reasons:

1. In the format: get Jovis("SaveFile","FileGlobal"), Jovis needs the global to know which file to save AND to get information as to how to report any errors.

2. In the format: get Jovis("SaveFile","FileGlobal", "All"), Jovis still needs the global in order to report errors AND to access the list of files in order to save them.

While the use of the global file identifier is different in either case, it is very much needed and should never be considered a "dummy" or unused variable in either situation.

Script Examples

To make referring to the "getFieldList" handler easier, we include it here once again :

```
function getFieldList listType
   if listType = "Fields" then
      return "First_Name,Last_Name,Is_Home_Address,Address1,Address2,City,"&
               "State,Zip,Phone,Customer_ID,Account_Start"
   else if listType = "Types" then
      return "text,text,logic,text,text,text,text,number,number,number,date"
   else if listType = "Indexes" then
      return "First_Name,Zip,Customer_ID"
   else if listType = "Sizes" then
      return "12,10,10"
   end if
end getFieldList
```

Error Messages That You May Encounter

Not a Valid Global
Not Enough parameters
Not a valid Jovis command
Invalid record
Missing or invalid fields
File opened read only
Field does not exist
Not enough memory

Client/Server version:
Record locked by another user
Record not locked

What You Should Have Learned

We put to use for the first time the commands 'CreateRecord', 'SetRecordField', 'UpdateRecord', and 'FillRecFields'. You should fully understand how each of these commands works in relation to each other. We also showed you a way to use the "AutoSave" feature in order to more quickly save records to disk.

What to learn next

Now that you understand how to create records, you are ready to learn how to retrieve records from a database file. In the next tutorial, we will introduce the 'ReadRecord', 'GetRecordField', 'LastRecord', 'NextRecord', 'PriorRecord', and 'CountMatches' commands. We will also cover how to delete a record using the 'DelRecord' command.

Tutorial 5

Retrieving Records

Before You Begin

This tutorial explains how to retrieve records from a data file. It requires that the "DASCO Database" be open and the relation "Customers" be setup. The record with the last name Smith must have been saved to the data file. The error and warning function handlers should already have been implemented.

Overview

In the previous tutorial we covered creating records and installing data into a record's fields. This Tutorial will cover retrieving records and getting information from specified fields in a record. We will also introduce the 'NextRecord', 'PriorRecord', 'LastRecord', and 'CountMatches' commands. Finally, we will show how to delete a record using the 'DelRecord' command.

Getting Started

ReadRecord

In our previous tutorial we demonstrated how to create a record, install the name "Smith" into the "Last_Name" field , and save the record to the data file. Now we will retrieve that same record:

```
1:    on RetrieveRecord
2:       put "Field Last_Name = [Smith]" into Criteria
3:       put Jovis("ReadRecord","myDB","Customers",Criteria) into aRecord
4:       put Jovis("GetRecordField","myDB",aRecord,"Customers","Last_Name")
5:                      into bg fld "Last_Name"
6:    end RetrieveRecord
```

Line 2 creates a criteria which tells Jovis to retrieve the first record whose "Last_Name" field equals "Smith". Keep in mind that there may be multiple records in which the "Last_Name equals "Smith", but this is the first record in the given relation that meets our criteria.

Line 4 uses the command called 'GetRecordField';it simply returns the data for the field referenced in parameter 4.

NEVER, under any circumstances, should you retrieve data from a Jovis record by setting the shell application's 'ItemDelimiter' property, and manually retrieving the field's data.)

In the above example, Jovis returns the field information, and puts it into a background field "Last_Name".   As you might suspect, there is a better way of doing this. This next example uses a repeat loop to populate the shell application's fields using 'GetRecordField' to access the record's fields.

```
on RetrieveRecord
   put "Field Last_Name = [Smith]" into Criteria
   put Jovis("ReadRecord","myDB","Customers",Criteria) into aRecord
   put getFieldList("Fields") into fldList
   repeat with x = 1 to number of items of fldList
      put Jovis("GetRecordField","myDB","Customers",item x of fldList)
                  into bg fld item x of fldList
   end repeat
end RetrieveRecord
```

The only drawback to this solution is that it depends on how fast the repeat loop can be executed by the shell application.

This final example of populating the shell application fields uses the 'FillHCFields' command. This is the fastest possible way to populate shell application fields. You should always use the 'FillHCFields' whenever you are displaying a record's information.

```
on RetrieveRecord
   put "Field Last_Name = [Smith]" into Criteria
   put Jovis("ReadRecord","myDB","Customers",Criteria) into aRecord
   put getFieldList("Fields") into fldList
   get Jovis("FillHCFields","myDB","Customers",aRecord,fldList,fldList)
end RetrieveRecord
```

Notice that we have named the shell application fields the same as the record's fields. This helps keep our database in parallel with out interface.

LastRecord

The 'LastRecord' command functions similarly to 'ReadRecord'. It takes a criteria and returns the last record that matches the given criteria. Once you have called 'LastRecord', you can use 'PriorRecord' and 'NextRecord' to move through the records that match the "current" criteria.

(Criteria will be discussed in tutorial seven.)

```
   put "Field Last_Name = [Smith]" into Criteria
   put Jovis("LastRecord","myDB","Customers",Criteria) into aRecord
```

NextRecord, PriorRecord

We mentioned earlier that our 'ReadRecord' example above returns the first record whose "Last_Name" field equals "Smith", and that there maybe additional records with "Smith" as the "Last_Name". By using the 'NextRecord' command we can easily get the next record that meets the "Last_Name" equals "Smith" criteria.

```
   put Jovis("NextRecord","myDB","Customers") into aRecord
```

If there are no further records, you will get a warning message from the 'JovisWarningMsg' function handler saying that "No more records satisfy the search criteria." Assuming that there were additional records, and you read the "next" record, you could then use the 'PriorRecord' command to "shift" back to the prior record. If you try to call 'PriorRecord' for the record that is before the first "Smith" record, you will get the same warning message about no further records matching the given criteria. As you have probably figured out, the 'ReadRecord' and 'LastRecord' commands control how 'NextRecord' and 'PriorRecord' respond. You must always "set" the criteria with 'ReadRecord' or 'LastRecord' before calling 'NextRecord' and 'PriorRecord'. In doing so, you establish what is called the "current" criteria.

CountMatches

'CountMatches' is a handy command for finding out how many records exist for a given criteria without having to step through the records and counting them. This command simply returns the record count for a criteria that you supply.

```
   put "Field Last_Name = [Smith]" into Criteria
   put Jovis("CountMatches","myDB","Customers",Criteria) into RecCount
```

DelRecord

Deleting records is a two step process:

1. Retrieve the record from the database, and

2. Call the 'DelRecord' command with the record that is to be deleted as the fourth parameter. Here is a complete example of what we mean:

```
      put "Field Last_Name = [Smith]" into Criteria
      put Jovis("ReadRecord","myDB","Customers",Criteria) into aRecord
      get Jovis("DelRecord","myDB","Customers",aRecord)
```

You will find that deleting a record is quite straight forward. We require that the record to be deleted is "sent" back to Jovis in parameter four, so there is no mistaking which record is going to be deleted. This is particularly important if you have multiple data files open with several relations in each file. (If you are using the client/server version of Jovis, you should consult the "Transactions" tutorial for details on using 'DelRecord'.)

Script Examples

To make referring to the "getFieldList" handler easier, we include it here once again.

```
   function getFieldList listType
      if listType = "Fields" then
         return "First_Name,Last_Name,Is_Home_Address,Address1,Address2,City,"&
                  "State,Zip,Phone,Customer_ID,Account_Start"
      else if listType = "Types" then
         return "text,text,logic,text,text,text,text,number,number,number,date"
      else if listType = "Indexes" then
         return "First_Name,Zip,Customer_ID"
      else if listType = "Sizes" then
         return "12,10,10"
   end getFieldList
```

Error Messages That You May Encounter

Errors:
Invalid input info
Not Enough params
Not a valid Jovis command
Not enough memory
No Record
Invalid record
Invalid field
Invalid operator
Missing operand
Missing field name
Missing background field name
Invalid background field name
Missing card field name

Warnings:
No more records satisfy search criteria
Record does not exist

What You Should Have Learned

We covered several commands in this tutorial. Most importantly, we discussed how to retrieve a record and populate the fields in the shell application's interface. We introduced the concept of a "current" criteria when using 'NextRecord' and 'PriorRecord' commands. The ability to find out how many records match a given criteria was also presented using the 'CountMatches' command. Finally, we showed you how to use the 'DelRecord' command to delete a record from the database.

What to learn next

Our next step is to create more records for our "Customers" relation. Rather than manually entering several dozen records for our tutorial, we will demonstrate how to use the 'ImportData' command. We will also show you how to export the data using the 'ExportData' command. Once we've imported our "data set", the subsequent tutorials will explain selections, record paths, and the 'Merge' command.

Tutorial 6

Importing and Exporting

Before You Begin

This tutorial explains how to import data from a text file into a Jovis database. We have provided a text file called "DASCO Customers.txt" with 25 records to be imported into our "DASCO" data file. You need to locate this file and have it ready for access. Your "DASCO" database file should be open and ready to be used.

Overview

The 'ImportData' and 'ExportData' commands provide the ability copy data to and from a Jovis data file. This is a fundamental capability for all databases, and allows you to process your data in more than a single application environment. For example, you might export the "DASCO" customer names and address to a word processor in order to perform a mail merge. In the other direction, the 'ImportData' command is a very fast way to enter multiple records from, say a customer list from a factory outlet store of "DASCO". Whatever the source or destination of your data, the 'ImportData' and 'ExportData' commands will be use repeatedly.

Getting Started

ImportData

We begin by using the 'ImportData' command to import 25 records from the text file called "DASCO Customers.txt" which is provided with this tutorial. Here is the script for importing these records.

```
on ImportToJovis
   put getFieldList("Fields") into fldList
   get Jovis("ImportData","myDB","Customers",fldList,tab,return)
end ImportToJovis
```

In line 2, our function handler 'getFieldList' returns our standard list of fields. The order of the fields must be the same order as the fields in the text file. (We have already provided them in order for this tutorial).

You do not have to import data for all of the relation's fields. For example, we could import records that do not include the 'Phone' field, in which case Jovis would leave the 'Phone' field empty in each record. It is important, however, that you are sure that the fields in the text file match and are in the same order as the list of fields being passed in parameter four of the 'ImportData' command.

Now we need to discuss the use of delimiters. Fortunately, Jovis allows you to use any delimiters you want for importing data. In our example above, we've used the shell application's 'tab' constant for the field delimiter and the 'return' constant for the record delimiter. You need to use delimiters wisely, because they must NOT be used within the fields that you are importing. If this happens, the data will be skewed across the newly created records, and you will be forced to trash your data file, and start all over. Because importing data affects your entire database file, you should always work with a backup copy. (Note that ASCII control characters 28, 29, and 30 are reserved for use by Jovis.)

If you are importing several thousand records, a progress dialog box will be displayed. There is a button in the dialog for aborting the import process. If you do abort, the integrity of your data file is left unstable. You will need to create a new data file, and re-import the data. (Obviously, aborting is meant as a last resort; it allows you to quickly go back to the beginning.)

If you have not already done so, you should now import the text file "DASCO Customers.txt" into your "DASCO Database". In our sample script above, we did not include the file path to the "DASCO Customers.txt" text file; as a consequence, Jovis will display the Macintosh 'Standard Get Dialog' so that you can select it. Once you have done this, Jovis will immediately import the information.

When the import is complete, you can use the "CountRelation' command we discussed in Tutorial 5 to find out how many records were created. Type the following into the message box.

```
   put Jovis("CountRelation","myDB","Customers")
```

(It should return 25.)

ExportData

We mentioned earlier that you may want to export your data for various uses, such as a mail merge. Here is an example that we could use to export data for our own mail merge:

```
1:    on ExportToFile
2:       put "First_Name,Last_Name,Address1,Address2,City,State,Zip" into
fldList
3:       put "Field Account_Start > [6/30/95]" into criteria
4:       get Jovis("ExportData","myDB","Customers",fldList,criteria,tab,return)
5:    end ExportToFile
```

In line 2, we specified a subset of fields from the "Customers" relation, and a local variable 'fldList' to contain the fields we need for a mail merge.
In line 3, we got a little creative, and supplied a criteria that indicates that we want to export all records whose "Account_Start" field is greater than "6/30/95". In other words, only customers who opened an account after June 30th, 1995 will be included. At this point, you should go ahead and export the data. The following 8 records and seven fields should have been exported:

First Name	Last Name	Address1	Address2	City	State	Zip
William	Anderson	11 West Cedar Street		Aurora	IL	60504
Susan	Abbado	NE Broker		Kennewick	WA	99337
Doris	Epstein	Prentice Place		Freeport	IL	61032
Judy	Ritter	Wine Road		Poughkeepsie	NY	12601
Barbara	Taylor	Bradford Lane		Penn Hills	PA	15123
Ruth	Addisen	1 North Express Drive	2nd Floor	Fort Bragg	CA	95437
Dorthy	Quinn	77 Running Brook		Moorhead	MI	56560
Mike	Stevens	Lancaster Street		Lansdowne	PA	19050

Script Examples

To make referring to the "getFieldList" handler easier, we include it here once again.

```
   function getFieldList listType
      if listType = "Fields" then
         return "First_Name,Last_Name,Is_Home_Address,Address1,Address2,City,"&
                  "State,Zip,Phone,Customer_ID,Account_Start"
      else if listType = "Types" then
         return "text,text,logic,text,text,text,text,number,number,number,date"
      else if listType = "Indexes" then
         return "First_Name,Zip,Customer_ID"
      else if listType = "Sizes" then
         return "12,10,10"
   end getFieldList
```

Error Messages That You May Encounter

Not a Jovis relational file
Not enough memory
Not enough parameters
Invalid number of import fields
Invalid field or record delimiter
Missing or invalid fields

What You Should Have Learned

This tutorial presented an introduction to the commands 'ImportData' and 'ExportData'. Along the way, we discussed the use of delimiters and why you should carefully use the delimiters that are NOT contained within your database. Also, you should keep in mind that Jovis reserves the use of the ASCII control characters 28, 29, and 30. We also used the 'ExportData' command in a "real world" example of exporting records to use for a mail merge.

Before you continue to a new tutorial, your "DASCO Database" file should now contain the 25 records from the "DASCO Customers.txt" text file.

What to learn next

Our next tutorial provides a full explanation of working with criteria, including range and compound criteria. Once you have learned about criteria, you will be ready to learn about selections and record paths.

Tutorial 7

Working with Search Criteria

Before You Begin

This tutorial explains how to worked with search criteria. You must have imported the data in the "DASCO Customers.txt" text file in order to use the examples in this tutorial. The error and warning function handlers are required as well. You will also need at least a few card buttons and a utility field into which you can dump the records that you retrieve.

Overview

This tutorial is devoted exclusively to showing you how to create both simple and complex criteria. Several of the Jovis commands include a parameter which requires criteria. In particular, 'ReadRecord', 'LastRecord', 'SetSelection', 'ExportData', 'AppendSelection', and 'TrimSelection' require criteria in order to retrieve records. Our goal is to teach you how to "build" you own criteria, and become familiar with this important capability.

The tutorial starts by introducing the basic elements of criteria. It continues with how to use these simple elements with the three operators 'and', 'not', and 'or' to build compound criteria. Next, we explain how to use indexes to optimize searches, as well as when optimization occurs. The tutorial includes many examples of what we have introduced. This tutorial is very important, so it is a good idea to try out the examples in your own scripts.

Getting Started

Jovis relational commands permit both simple and complex search criteria. All of the commands using search criteria require the same format. Search criteria normally consist of an operand followed by an operator followed by another operand. This is called a "Term". For example:

```
Field Last_Name = [Smith]
```

The operator in the above example is the equals sign ("="). The left operand is "Field

Last_Name", and the right operand is "[Smith]". Note that you must use either quotes or square brackets for the right operand. Both of the following criteria are acceptable:

```
   put "Field Last_Name = [Smith]" into aCriteria
   put "Field Last_Name = " & quote & "Smith" & quote into aCriteria
```

The left operand must be the name of a Jovis database field, such as "Zip", "Last_Name", or "Customer_ID". The right operand can be a shell application constant, a literal string, or a local or global variable.

Operators can be any of the following:

```
   =            Left Operand equals Right Operand
   is           Left Operand equals Right Operand
   ≠            Left Operand does not equal Right Operand
   <>           Left Operand does not equal Right Operand
   >            Left Operand is greater than Right Operand
   >=           Left Operand is greater than or equal to Right Operand
   ≥            Left Operand is greater than or equal to Right Operand
   <            Left Operand is less than Right Operand
   <=           Left Operand is less than or equal to Right Operand
   ≤            Left Operand is less than or equal to Right Operand
   in           Left Operand is contained in Right Operand
   contains     Left Operand contains Right Operand
   starts_with  Left Operand starts with Right Operand
```

The type of comparison (numeric, date, logic or text) being done depends upon the field type for the database field.

The operators “in”, “contains”, and “starts_with” will always return false if the field type is NOT text. Case is ignored when comparing text fields, however diacriticals are not ignored. For example, the following are some VALID expressions:

```
on createCriteria
   global gCustomer_ID

   put "field last_name = [smith]" into myCriteria

   put "field zip >= [" & bg fld "zip" & "]" into myCriteria
   -- or:
   put "field zip ≥ " & quote & bg fld "zip" & quote into myCriteria

   put "field Last_Name starts_with [smi]" into myCriteria

   put "Field Customer_ID = [" & gCustomer_ID & "]" into myCriteria

end createCriteria
```

These are INVALID expressions and the reasons they are invalid:

```
field "zip" > [60007]      -- do not use quotes around field name in left operand

field customer_ID => "47"  -- '=>' is not a valid operator; it should be >= or ≥

field customer_ID ≥ 47     -- need quotes or brackets around constant value
```

For most of the operators, you do not need to use spaces to separate the operands in your criteria. The exceptions are the operators not, is, in, contains, and starts_with.

For example, all of these criteria are equivalent:

```
Zip  =  "10010"
Zip ="10010"
Zip= "10010"

Last_Name starts_with [Jack]   -- space is required between "Last_Name" and "Starts_with"
```

NOTE: You will notice that several of the Jovis commands allow you to optionally omit the criteria and leave the parameter empty. If you do decide to do this, the default criteria is:

```
Field !RecID > [0]
```

Compound Criteria

A single comparison is often too limiting. In many cases you will want to select records that satisfy more than one criterion.

Jovis allows this by letting you take terms, such as those described above, and combine them with the operators (not, and, or) to develop compound search expressions.

For example, if you wanted to select customers whose last name is "Smith" and who live in "Connecticut", you would use this expression:

```
Field Last_Name = "Smith" and Field state = "CT"
```

The order in which an expression is evaluated is important. Let us consider the results of 2 expressions using the following five records:

Rec#	Name	Zip
1	Jones	60615
2	Smith	40412
3	Jones	60235
4	Smith	60007
5	Smith	74715

For example, look at the following expression.

```
   Field Last_Name = "smith" or Field zip >= "60000" and Field zip <= "69999"
```

Jovis evaluates expressions from left to right. In this case, it first retrieves all records where "Last_Name" is "Smith" or the "Zip" is greater than 60000, then from that group, it selects a subset of records where the zip is less than 69999.

Using this expression, record #2 is not selected because the zip code is less than 60000, and record # 5 is not selected since the zip code is greater than 69999, even though the name is Smith in both records.

Let’s change the above expression to select every record whose zip code is in the range 60000 to 69999 or whose last name is Smith. You can either change the order of the terms, or use parentheses, as follows:

```
   Field Last_Name = "smith" or (Field zip >= "60000" and Field zip <= "69999")
```

Using this second expression, all five records are selected, since either the name is Smith or the zip code is in the range 60000 to 69999.

Let's take a different example of search criteria, using the following six records:

Rec #	Company Name	Position Title
1	Commonwealth Co.	Personnel Manager
2	U.S. Government	Manager of Employee Relations
3	U.S. Government	Personnel Manager
4	General Data	Director of Personnel
5	Computers etc.	Personnel Specialist
6	Commonwealth Co.	Vice President of Development

Try this search criteria on the above records:

```
   Field company_name <> "U. S. Government" and
   (Field position_title contains "personnel" or
   Field position_title contains "employee relations")
```

Since Jovis evaluates expressions from left to right, it first selects all records where the company name is not "U. S. Government". From that group it then selects any records where the position title contains either the word personnel or the words employee relations. From the six records shown above, the records selected are 1, 4, and 5. Notice that the operators “in” and “contains” are opposites. Instead of writing:

```
   put "Field position_title contains [personnel]" into criteria
```

you could have written:

```
   put "[personnel] in Field position_title" into criteria
```

and obtained the same results.

Optimizing Searches

A search is optimized by limiting the number of records that must be read during the search process. For example, if you want to find everyone named Smith who lives in Connecticut, you can optimize the search by only looking at records for people named Smith, and from that group, those who live in Connecticut. Or you can read only the records of people who live in Connecticut and then select from that group those people whose name is Smith.

This “narrowing down” process is done by using indexes. An index is simply a field that is sorted separately from the record in order to allow fast searches. In the example above, you might use the following search criteria:

```
   Field Last_Name = "Smith" and Field state = "CT"
```

If the field State is indexed, the database goes directly to those records where the State field is "CT". If the State field is not indexed, but the Last_Name field is indexed, the database goes directly to the records of people whose last name is "Smith".

Jovis optimizes on the first indexed field that it finds. Since it evaluates the search criteria from left to right, the order in which the search terms are presented directly affects the speed in which the records are found.

In the example above, if both fields are indexed, Jovis uses the Last_Name index. If there are many more people named "Smith" than people who live in the state "CT", the search could be improved by listing the state field first, like this:

```
   Field State = "CT" and Field Last_Name = "Smith"
```

Search criteria can only be optimized for a field if:

(1) the values of the field constitute a range,
(2) the field is indexed, and
(3) the succeeding terms are “and”ed to the range terms.

(1) A range can be any of the following combinations:

```
Field A  =           "value"
Field A  starts_with "value"
Field A  >           "value"
Field A  >=          "value"
Field A  <           "value"
Field A  <=          "value"
Field A  >           "value1" and Field A <  "value2"
Field A  >=          "value1" and Field A <  "value2"
Field A  >           "value1" and Field A <= "value2"
Field A  >=          "value1" and Field A <= "value2"
```

Notice that the operators ≠ , < >, in, and contains cannot be optimized.

(2) In all of the above cases, database field A must be an indexed field.

(3) This criteria is said to be “and”ed with the range.

```
   Field A = "value1" and Field B ≠ "value2"
```

In this case, Field A = “value1” is a range. Field B ≠ “value2” must be true in addition to the Field A's equality, therefore this search is optimized.

On the other hand, if we change the above criteria to:

```
   Field A = "value1" or Field B ≠ "value2"
```

This search cannot be optimized, because the first term does constitute a range, and, it is not combined with the second term using an “and” operator.

There are times when it may not make sense to index a field. Since the purpose of an index is to speed up a search, the possible number of duplicate values in the indexed field is something to think about. A logic field, in particular, has only two possible values - true or false, and so will always have many duplicates. Indexing on a logic field for index searching is rarely beneficial, since the database will end up scanning many records along the index to find the one where a given field is, say, true and "Last_Name equals "Smith".

To summarize, some fields have just too many duplicate values to make useful indexes. Logic fields especially are much better used as a qualifying term “and”ed to the criteria, as below.

```
Jovis last_name = "Smith" and Jovis Account_Start = "true"
```

Now let us take a look at how to optimize an expression with many search criteria.

```
Field zip > "59999" and
Field zip <= "69999"  and
(Jovis last_name = "Smith"  and
   (Field phone contains "312" or
      Field phone contains "708" or
      Field phone contains "815"))
```

The first part of the expression constitutes a range, and the zip code field is indexed, so Jovis uses this index to first select all the records between 60000 and 69999.

Jovis then looks within that selection for records with "Last_Name" = "Smith" and with the specified numbers within the phone number field.

Notice that even if we reverse the first two terms, Jovis still recognizes it as a range and optimizes on the zip code field. For example:

```
Field zip <= "69999"  and
Field zip > "59999" and
(Jovis last_name = "Smith"  and
   (Field phone contains "312" or
      Field phone contains "708" or
      Field phone contains "815"))
```

You could probably make the search even more efficient by optimizing on the name field, rearranging the terms within the expression like this:

```
Field last_name = "Smith"  and
(Field zip > "59999" and Field zip <= "69999") and
   (Field phone contains "312" or
      Field phone contains "708" or
      Field phone contains "815")
```

This is a better way to search if there are fewer Smiths than there are records within the zip code range, and if "Last_Name" is an indexed field.

Changing the expression again, see what happens in this situation:

```
(Field phone contains "312" or
   Field phone contains "708" or
   Field phone contains "815") and
Field last_name = "Smith" and
Field zip > "59999" and Field zip <= "69999"
```

Remember that the operator “contains” cannot be optimized. Therefore, the first range that Jovis encounters is still Last_name = Smith.

One way NOT to order the first three terms would be:

```
Field zip <= "69999" and
Field last_name = "Smith" and Field zip > "59999" and
(Field phone contains "312" or
   Field phone contains "708" or
   Field phone contains "815")
```

Doing it this way is disastrous! Jovis optimizes on the first range, which is zip code < = 69999, so it first gets ALL the records with zip code 00000 - 69999. Then it finds all the Smiths and zip codes greater than 59999, and finally looks at the phone numbers.

Script Examples

Here are some more examples of criteria for use with our "DASCO" database:

```
Field last_name starts_with [s] and Field Account_Start > "6/30/95"

Field cusomter_ID = [4761]

Field Account_Start ≥ "6/30/95" and Field Account_Start ≤ "6/30/96"

Field state = "IL" and Field Account_Start > "6/30/95"
```

Error Messages That You May Encounter

Warnings:
No more records satisfy criteria
Missing or invalid fields
Selection exceeds 30000
Record does not exist
No records selected

Errors:
Invalid selection name
Invalid operator
Missing operand
Unbalanced quotes or brackets

Missing field name
Invalid Selection field name
Incorrectly placed left parenthesis
Unbalanced parenthesis
Incorrect placed NOT operator
Incorrect placed AND operator
Incorrect placed OR operator
One of the operands must be a field

What You Should Have Learned

You should be able to create at least simple criteria without referring to this tutorial or other examples at this point. Because compound criteria is more complex, referring to the documentation may be required. You should feel comfortable with using all of the operators, and clearly understand the difference between a left and right operand. On occasion, it is worth reviewing this tutorial, so you do not forget some of the details that were covered.

What to learn next

Now that we have covered creating criteria, we will move on to creating selections or subsets of a relation. Selections are one of the most powerful features in Jovis. Once you have started creating selections on your own, you will have completed the core or basic tutorials.

The remaining tutorials explain "peripheral" features, such as record paths, multiple sections, and the 'Merge' command. We are sure that sooner or later you will need to understand and use these capabilities, but they are not crucial to getting started with Jovis.

Tutorial 8

Creating Selections

Before You Begin

This tutorial explains how to create selections. You will need to have your "DASCO Database" open, and you should have already imported the "DASCO Customers.txt" text file. If you have not yet studied the tutorial called "Working with Criteria," you should do that now. This tutorial is the last of the eight "core" tutorials which cover the basic features of Jovis. (If you are using the client/server version of Jovis, you need to study the tutorial on "Transactions".)

Overview

Creating a selection is a basic requirement of all relational databases. We think you will be impressed with how powerful and flexible this feature is in Jovis.

A selection is a subset of a relation. It has columns and rows just like a relation. The major difference between a selection and a relation is that a selection is "read-only"; you cannot make changes to a selection. You access records through a selection, make changes to the records, and then save the records back to the database. Selections can have as many fields or columns as are available in a given relation, and as many rows as the provided criteria will generate. One nice feature in Jovis is that you can "save" a selection to a text file, using the 'ExportSelection' command. You should keep in mind that selections are held in memory. (In the client/server version, they are held in memory at the client.). The amount of memory allocated to the shell application directly effects the size and number of selections you can work with.

The first command to be introduced will be 'SetSelection', followed by 'SelectionToVar', 'SortSelection', 'GetSelectionRecord', and 'ExportSelection'. We will also show you how some of the commands we have previously introduced are used to support creating and working with selections.

Getting Started

You use the 'SetSelection' command to create a selection. This generates the selection from the given relation based on the criteria you have provided. When you create a new selection for a given relation, the previous selection for that relation is removed from memory. You can create a current selection for each relation and access it at anytime. Once the selection is created, you can put it into a variable or a shell application field using the command 'SelectionToVar'. For example:

```
   on CreateSelection
      put "Field Customer_ID > [0]" into aCriteria
      put "First_Name,Last_Name,Address1,Address2" into fldList
      get Jovis("SetSelection","myDB","Customers",fldList,aCriteria)
      put Jovis("SelectionToVar","myDB","Customers",comma,return)
                     into cd fld "Display"
   end CreateSelection
```

Once the selection has been created, we use the 'SelectionToVar' command to put the selection into card field "Display". Notice that we are using the comma, which is shell application constant, as a field delimiter, and a carriage return (also a shell application constant) as the record delimiter. Internally, Jovis uses its own delimiters for the selection; however, you can specify any character that you wish as a delimiter. Just be sure that the characters you use for delimiters are not used in the data you have selected. If this happens, the fields and records well be skewed when you display them.

Many users of Jovis use our product "ListTable™" to display their selections in a "spreadsheet" style window. ListTable provides full scripting access to each record's field, and makes it very easy to not only view your selections, but to make changes to the database. ListTable does not have a 32k text limit, so if your selections are greater than 32k, ListTable will still be able to display them.

CountSelection

If you have already implemented the above example, you already know that it selects all the records whose "Customer_ID" field is greater than zero. Because all of our records have a "Customer_ID" greater than zero, all the records in the database will be selected. Notice that the we are only selecting the first name, last name, and customer ID; all the other fields in the relation are omitted. If you need to know how many records were selected, you can use the 'CountSelection' command, like this:

```
put Jovis("CountSelection","myDB","Customers") into SelectionCount
```

SortSelection

Selections can be sorted by any of the fields in the selection. They can also be sorted on more than one field. For example, we could sort the selection by last name, and then sort it by first name, like this:

get Jovis("SortSelection","myDB","Customers","Last_Name","First_Name")

This will give us, for example, all of the "Smith" records grouped together, sorted by first name within that group.

GetSelectionRecord

Now that we have created a selection, we may want to "update" some information for a given customer. Let's say that "Alice Brown", which is row number 3 in the selection, has changed her address. It was "RR 1", and we want to change it to "House #123". Here is the script that makes this change:

```
on UpdateSelectionRecord
   put Jovis("GetSelectionRecord","myDB","Customers",3) into aRecord
   put Jovis("SetRecordField","myDB","Customers",
                  aRecord,"Address1","House #123") into aRecord
   get Jovis("UpdateRecord","myDB","Customers",aRecord)
end UpdateSelectionRecord
```

You can see how the commands that we introduced in the "Create Record" tutorial come in handy here. We retrieve the third record using the 'GetSelectionRecord' command, and then use the 'SetRecordField' and 'UpdateRecord' commands to make and save our changes.

ExportSelection

We mentioned earlier that you can save a selection to a text file. Using the 'ExportSelection' command, this is an easy process. Here's an example:

```
   on SelectionToFile
      put "Field Customer_ID > [0]" into aCriteria
      put "First_Name,Last_Name,Address1,Address2,City,State,Zip" into fldList
      get Jovis("SetSelection","myDB","Customers",fldList,aCriteria)
      put Jovis("ExportSelection","myDB","Customers",tab,return)
                     into cd fld "Display"
   end SelectionToFile
```

The 'ExportSelection' command has several optional parameters. In our example, because we didn't provide a file path for creating the text file, the Macintosh "Standard Put File" dialog will be displayed. Once the file is saved, you could use it to create a mail merge to the selected customers.

Script Examples

Here is another example that creates a selection and displays it in a shell application's field called "Display".

```
on CreateSelection
   put "Field last_name starts_with [s] and
         Field Account_Start > [3/31/94]" into aCriteria
   put "First_Name,Last_Name,Account_Start,Customer_ID" into fldList
   get Jovis("SetSelection","myDB","Customers",fldList,aCriteria)
   get Jovis("SortSelection","myDB","Customers","Account_Start")
   put Jovis("SelectionToVar","myDB","Customers",comma,return)
                  into cd fld "Display"
   put "Row count: " & Jovis("CountSelection","myDB","Customers") into msg
end CreateSelection
```

Error Messages That You May Encounter

Warnings

No more records satisfy criteria
Selection exceeds 30000
No records selected

Errors

Field does not exist in relation
Invalid field
Not Enough Memory
Not a valid relation name

What You Should Have Learned

You should have learned how to create and display a selection. We introduced the 'CountSelection' and 'SortSelection' commands. In addition, we showed you how to change data by referring to the row number in the selection and using 'SetRecordField' and 'UpdateRecord' to save your changes. We also covered how to export a selection to a text file using the 'ExportSelection' command.

What to learn next

You have been working with and creating selections which are the current or active selection for a given relation. When you create a new selection, the old one is removed from memory. In the next tutorial, we explain how to reserve selections by assigning them a name and "deactivating" them. This allows you to swap selections in memory without re-creating them. This is a very useful feature for applications which need to support an "undo" capability.

Tutorial 9

Multiple Selections

Before You Begin

This tutorial explains the additional capabilities that you can use when creating selections. You need to know how to work with criteria and you should feel comfortable creating selections with little difficulty. Once again we will use our "DASCO Database", so you will need to have it open and ready to go. The warning and error function handlers should be in place so that you can catch any problems as they occur.

Overview

The four commands 'ReserveSelection', 'RestoreSelection', 'ListSelections', and 'DelSelection' are used with multiple selections. These commands allow you to "set aside" the current selection for a given relation and create another selection in that same relation. Keep in mind that for each relation, there can only be a single current or active selection. By reserving and restoring a selection you can control which selection you want to "currently" work with. As with all selections, you should keep in mind that selections are held in memory, and that the amount of memory allocated to the shell application affects the size and number of selections you can work with.

The basic process for reserving a current selection is to call 'ReserveSelection', and provide a name for the current selection. Once the 'ReserveSelection' command has been successfully called, the current selection becomes a named selection, so there is no longer a current selection. (You could, as an alternative, use the 'DupSelection' command to duplicate the current selection without clearing it.)

With the 'RestoreSelection' command, you cause a named selection to become the current selection. In doing so, you will clear the current selection, if there is one. Most of the other selection commands such as 'SelectionToVar', 'CountSelection', and 'GetSelectionRecord' have an optional parameter for a named selection. By using this parameter, you can access a "reserved" selection without making it the current selection. (Note that this does not apply to the 'SortSelection' command.)

Getting Started

In order to reserve a current selection you must include a unique name in the fourth parameter of the 'ReserveSelection' command. The current selection must be a valid selection with at least one record selected. You can use the 'ListSelection' command to get a list of the named selections for the given relation. This is handy for testing if a named selection already exists. The 'RestoreSelection' command clears the current selection and installs the named selection and removes it from the list of reserved selections. If you do not need the reserved selection for further processing, you should clear it from memory using the 'DelSelection' command. Here's an example using all four of these commands:

```
on UseMultiSelections
   put "Field State starts_with [Il]" into aCriteria
   put "First_Name,Last_Name,Account_Start,Customer_ID" into fldList
   get Jovis("SetSelection","myDB","Customers",fldList,aCriteria)
   if Jovis("CountSelection","myDB","Customers") > 0 then
      get Jovis("ReserveSelection","myDB","Customers","Illinois_Customers")
   end if
   --
   put "Field State starts_with [VT]" into aCriteria
   get Jovis("SetSelection","myDB","Customers",fldList,aCriteria)
   if Jovis("CountSelection","myDB","Customers") > 0 then
         get Jovis("ReserveSelection","myDB","Customers","Vermont_Customers")
   end if
   --
   answer "The currently reserved selections are:" & return & return &
                get Jovis("ListSelections","myDB","Customers")
   --
   if "Illinois_Customers" is in Jovis("ListSelections","myDB","Customers") then
      get Jovis("RestoreSelection","myDB","Customers","Illinois_Customers")
      put Jovis("SelectionToVar","myDB","Customers",comma,return) into
         cd fld "MidWest Customers"
      get Jovis("ReserveSelection","myDB","Customers","Illinois_Customers")
      get Jovis("DelSelection","myDB","Customers","Illinois_Customers")
   end if
   --
   if "Vermont_Customers" is in Jovis("ListSelections","myDB","Customers") then
      get Jovis("RestoreSelection","myDB","Customers","Vermont_Customers")
      put Jovis("SelectionToVar","myDB","Customers",comma,return) into
         cd fld "NorthEast Customers"
      get Jovis("ReserveSelection","myDB","Customers","Vermont_Customers")
      get Jovis("DelSelection","myDB","Customers","Vermont_Customers")
   end if
end UseMultiSelections
```

While this is a rather long example, you can easily see that all we have done is create two selections, one called, "Illinois_Customers" and one called "Vermont_Customers", and reserved them using the "ReserveSelection' command. Once that was done, we restored each selection, and then, using the 'SelectionToVar' command, we put the selections into their respective shell application fields. You probably noticed the numerous "if" statements in the example. Careful testing for the existence of a named selection, and "if" the 'SetSelection' command successfully created a selection, keeps our program flowing without unnecessary warnings and errors.

Script Examples

The "QuickSelection" handler below shows how you might implement the two following utility handlers, "saveCurrentSelection" and "RestorePriorSelection", which can be used to implement a simple selection "Undo". Basically, you reserve the current selection, and make another selection. If this new selection comes back empty, you can then revert back to the previous selection, providing a type of undo process.

```
on QuickSelection
   saveCurrentSelection
   put "Field Customer_ID > [0]" into aCriteria
   put "First_Name,Last_Name,Address1,Address2" into fldList
   get Jovis("SetSelection","myDB","Customers",fldList,aCriteria)
   if Jovis("CountSelection","myDB","Customers") > 0) then
      put Jovis("SelectionToVar","myDB","Customers",comma,return)
                  into cd fld "Display"
   else
      RestorePriorSelection
   end if
end QuickSelection

on saveCurrentSelection
   if Jovis("countSelection","myDB","Customers") > 0 then
      if "MyPriorSelection" is in Jovis("listSelections","myDB","Customers") then
         get Jovis("delSelection","myDB","Customers","MyPriorSelection")
      end if
      get Jovis("reserveSelection","myDB","Customers","MyPriorSelection")
   end if
end saveCurrentSelection

on RestorePriorSelection
   if "MyPriorSelection" is in Jovis("listSelections","myDB","Customers") then
      get Jovis("restoreSelection","myDB","Customers","MyPriorSelection")
   end if
end RestorePriorSelection
```

Error Messages That You May Encounter

Not Enough Memory
Not a valid relation name
Selection name already exists

What You Should Have Learned

In this tutorial we demonstrated how to work with the multiple selection commands. By using 'ReserveSelection' you can name and reserve the current selection for the given relation. When you want to access this selection you can use the 'RestoreSelection' command. In addition, we explained that the current selection is cleared when 'ReservedSelection' is called. This sets the stage for either creating a new selection or restoring a named selection using the 'RestoreSelection' command.

What to learn Next

Working with multiple selections is quite similar to working with record paths, so we have placed the "Record Paths" tutorial next. What record paths are and how you can use them will be explained. You will find that many of the same concepts used with multiple selection are also applied to record paths, particularly that of reserving a record path by name, and then restoring it for current use.

Tutorial 10

Record Paths

Before You Begin

The concept of using "Record Paths" is similar to using bookmarks; by reserving a record path, you can return to it at a later time. Record paths can be very useful, especially if you need to refer to different records within the same relation.

This tutorial assumes that you understand how to use the 'ReadRecord', 'NextRecord', 'PriorRecord' and 'LastRecord' commands, and have studied the tutorial "Retrieving Records," which explains how to use these commands. As with all of our tutorials, this one uses the "DASCO Database" file for the examples we will be presenting.

Overview

As you use the 'ReadRecord', 'NextRecord', 'PriorRecord' and 'LastRecord' commands, you will find situations when you need to access a record in the same relation using a criteria different than the current one. With the 'ReserveRecordPath' command, you can store your position in the database, as well as the current criteria, in memory. In doing so, Jovis clears the current position and criteria. Calling 'ReadRecord' at this point would result in an error.

To return to the reserved record path, you use the 'RestoreRecordPath' command, which clears the current record path and criteria, and installs the reserved record path. It also removes it from the list of reserved record paths. The current record is now the record you last read using one of the 'ReadRecord' commands. You can test this by using the 'CurrentRecord' command. In addition to the 'ReserveRecordPath' and 'RestoreRecordPath' commands, there are the commands 'ListRecordPaths', 'CurrentRecord', 'DupRecordPath', and 'DelRecordPath' which we will also discuss in this tutorial.

Getting Started

When you call the 'ReserveRecordPath' command, you provide a unique name for the reserved record path. The 'ListRecordPaths' will provide you with a list of the reserved record paths.

'DelRecordPath', of course, deletes the named record path. The following example is a utility routine that checks a customer has not been entered a second time subsequent to this or her "Account_Start" date.

```
on IsDuplicateCustomer
   put "Field Customer_ID = [4432]" into aCriteria
   put Jovis("ReadRecord","myDB","Customers",aCriteria) into aRecord
   put Jovis("GetRecordField","myDB","Customers",aRecord,"Last_Name") into LastName
   put Jovis("GetRecordField","myDB","Customers",aRecord,"Account_Start") into
                  AccountStart
   get Jovis("ReserveRecordPath","myDB","Customers","myRecordPath")
   put "Field Last_Name = [" & LastName & "] and Field Account_Start > [" &
                  AccountStart &"]" into tempCriteria
   put Jovis("ReadRecord","myDB","Customers",tempCriteria) into tempRecord
   if tempRecord is not empty then
      answer "Found possible duplicate record."
      if "myRecordPath" is in Jovis("ListRecordPaths","myDB","Customers") then
         get Jovis("DelRecordPath","myDB","Customers","myRecordPath")
      end if
      exit IsDuplicateCustomer
   else
      get Jovis("RestoreRecordPath","myDB","Customers","myRecordPath")
      put Jovis("CurrentRecord","myDB","Customers") into aRecord
      put Jovis("NextRecord","myDB","Customers",aCriteria) into aRecord
      -- etc.
   end if
end IsDuplicateCustomer
```

This routine calls 'ReadRecord' and gets the last name and account start field information. It then calls 'ReserveRecordPath' to save our location in the database. Next we create a criteria using the last name and account start information and make a second 'ReadRecord' call. If we find another record with the same last name, and whose account start field is greater than our original account start field, then we might suspect that a customer's account has been entered twice. If we do not find a duplicate account, we call 'RestoreRecordPath,' which returns us to exactly where we left off.

After restoring the record path, we called 'CurrentRecord' just to show that we are back where we left off. From here you can continue with further processing, or return to the calling handler.

Script Examples

The utility handlers "saveCurrentRecPath" and "RestorePriorRecPath" provided below, can be used to implement a simple read record undo. Our "QuickReadRec " handler shows how you might implement these utility handlers. Basically, you reserve the current record position, and make another 'ReadRecord' call, and if this new 'ReadRecord' call comes back empty, you revert back to the previous record location.

```
on QuickReadRec Cust_ID
   saveCurrentRecPath
   put "Field Customer_ID = [" & Cust_ID & "]" into aCriteria
   put "First_Name,Last_Name,Address1,Address2" into fldList
   put Jovis("ReadRecord","myDB","Customers",fldList,aCriteria) into aRecord
   if aRecord is not empty then
      put Jovis("GetRecordField","myDB","Customers",aRecord,"Last_Name")
                  into cd fld "Display"
   else
      RestorePriorRecPath
   end if
end QuickReadRec

on saveCurrentRecPath
   if "MyPriorRecPath" is in Jovis("ListRecordPaths","myDB","Customers") then
         get Jovis("DelRecordPath","myDB","Customers","MyPriorRecPath")
      get Jovis("ReserveRecordPath","myDB","Customers","MyPriorRecPath")
   end if
end saveCurrentRecPath

on RestorePriorRecPath
   if "MyPriorRecPath" is in Jovis("ListRecordPaths","myDB","Customers") then
      get Jovis("RestoreRecordPath","myDB","Customers","MyPriorRecPath")
   end if
end RestorePriorRecPath
```

Error Messages That You May Encounter

Invalid character
Reserved record path already exists
Not enough parameters
No current record
Reserved record path not found

What You Should Have Learned

We introduced the concept of "Record Paths" which is similar to bookmarks. You should have learned how to reserve and restore a location in a relation. In addition, we covered how to delete a reserved record path as well as get a list of reserved record paths.

What to learn Next

Both the single-user and client/server versions of Jovis use the same database file format and set of commands. Therefore, it is easy to switch between the single and multi-user versions. In the next tutorial, we introduce the concept of "Transactions". This allows a client to "lock" a record in order to make changes, and at the same time, prevent other clients from making changes to the same record. If you are using the single-user version, and know you might be using the client/server version in the future, it is a good idea to script in the transaction commands. Jovis ignores these commands under the single-user version; no errors will be generated. Therefore, you can script for the client/server version within the single-user version, and your application will work correctly in both versions.

Tutorial 11

Transactions

Before You Begin

In this tutorial we introduce the "Transaction" commands. These commands are used for locking records, and thereby preventing more than one client from making changes to a record at the same time. Even if you are using the single-user version, it is important that you consider scripting the "Transaction" commands; then, if you should begin using the client/server version, your scripts will be ready to go. It is always easier to script future needs in while you are designing, typing, and testing than to retro-fit a new layer of commands into what you have already done.

You need to have learned how to use the read commands, 'ReadRecord', 'NextRecord', 'PriorRecord', 'LastRecord', and 'GetSelectionRecord', as well as the 'CreateRecord', 'DelRecord', and 'UpdateRecord' commands. These are the commands that are used in conjunction with the "Transaction" commands. You must be sure that the warning and function handlers have been installed. They become invaluable when transaction errors occur. In addition, you should have the "DASCO Database" open and ready for use.

Overview

The transaction commands are necessary for locking records before updating in a multi-user situation. This requirement is to prevent two clients from updating the same record at the same time.

In the single-user version, the transaction commands do nothing and return no errors; they have no effect in a single-user script. This allows you to write scripts that work with both the single and client/server versions.

We will spend time not only showing you how to use these commands, but also how best to implement them into the flow of your applications.

Getting Started

When a script is about to read a record or series of records which will be updated or deleted, you must first call the 'BeginTransaction' command before calling 'ReadRecord', or any of the other "read" commands.

Once you have called 'BeginTransaction', any records read by the script are locked by the Server, so that other clients on the network will be able to look at the records, but will not be able to "update" them. Other clients trying to lock the records will get the error message, "Record locked by another".

When the client who locked the record makes a change, updates, and reads the record again before committing the transaction, he or she will see the record with the changes that have been made. At the same time, other clients reading the record will see the record as it was prior to the changes.

The transaction commands work at the collection level of your database, NOT at the relation level. In other words, the 'BeginTransaction' command will affect all the relations within a given collection or database file. You must call 'BeginTransaction' for each data file.

The Server limits the number of records in each relation that may be locked at one time. For example, a basic 3-client Server allows a maximum of 120 locked records per relation at one time by all of the connected clients. If another 3 clients are added to the Server, the limit is increased to 240 records per relation for all of the clients. This limit is intended to keep collisions among clients trying to update the same records to a minimum. When scripting transactions, you should think about the client accessing and updating relevant records, and committing the transaction so that the records are released to other users of the database. You should always try to keep to a minimum the number of records being read after you've called 'BeginTransaction'.

The 'UpdateRecord' and 'DelRecord' commands do not physically change the database until the script executes the 'CommitTransaction' command. At that time, any changes made to the records are applied to the database, and all records locked by the 'BeginTransaction' command are unlocked.

If the script executes the 'CancelTransaction' command instead of 'CommitTransaction', then all the changes are thrown away, and all the records locked by the 'BeginTransaction' command are unlocked.

As mentioned before, if a script needs to lock records contained in more than one database file, it must execute the transaction commands for each database file.

When new records are being added to the database, the commands 'CreateRecord' and 'UpdateRecord' should also be preceded by transaction commands.

It is impossible to update or delete a record without first locking it. If a script tries to use either of these commands outside a transaction, the error message, “Record not locked” is returned.

Records in a selection are never locked, since they are actually a “snapshot” of the records themselves. So it makes no difference if one client has locked some of the records that go into a selection being made by another client.

Script Examples

Here is a basic example of using the transaction commands to update a record:

```
1:    on ChangeRecordField
2:       if Jovis("IsTransactionOn","myDB") = "False" then
3:          get Jovis("BeginTransaction","myDB")
4:       end if
5:       put "Field Customer_ID = [4432]" into aCriteria
6:       put Jovis("ReadRecord","myDB","Customers",aCriteria) into aRecord
7:       put Jovis("SetRecordField","myDB","Customers",aRecord"Address1",
8:                "PO Box 55") into aRecord
9:       get Jovis("UpdateRecord","myDB","Customers",aRecord)
10:      get Jovis("CommitTransaction","myDB")
11:   end ChangeRecordField
```

The above example will run exactly the same whether you are using the single or client/server version of Jovis. By calling the 'IsTransactionOn' command we can avoid a possible error if the "transaction flag" at the server has already been set. (The 'IsTransactionOn' command will always return "False" in the single-user version.) Lines 6 though 9 should be familiar; we simply retrieve a record from the server, change the "Address1" field and call 'UpdateRecord'. The 'CommitTransaction', in line 10, tells the server to commit the record to the database file.

The following example demonstrates the 'CancelTransaction' command. (Notice that we do not call 'UpdateRecord' in order to delete a record.) Once 'DelRecord' has been called, the 'CommitTransaction' command will do the actual deletion of the record from the database.

Script Examples

```
      on DeleteRecordField
      if Jovis("IsTransactionOn","myDB") = "False" then
         get Jovis("BeginTransaction","myDB")
      end if
      put "Field Customer_ID = [4432]" into aCriteria
      put Jovis("ReadRecord","myDB","Customers",aCriteria) into aRecord
      if aRecord is empty then
         get Jovis("CancelTransaction")
      else
         get Jovis("DelRecord","myDB","Customers",aRecord)
         get Jovis("CommitTransaction","myDB")
      end if
   end DeleteRecordField
```

Error Messages That You May Encounter

Errors:
Record not locked
Max locks for file reached
Record locked by another user
Server not responding
File open read only

Warnings:
Transaction already on
Transaction not on

What You Should Have Learned

We covered a lot of material in this tutorial, and you may want to review it on occasion. In particular, you should have learned that 'BeginTransaction' must be called before any of the read commands, and that the 'CommitTransaction' command does the actual updating, creating, or deleting that you require. We explained about the maximum number of records per relation that can be locked, and that locked records can only be changed by the client who initiated the transaction. We also explained that the transaction commands can be scripted into the single-user version and will not cause an error. This allows for compatibility with the client/server version. Finally, we provided two scripting examples on which you can model your own applications.

What to learn Next

Our next tutorial explains the Merge command, which provides the ability to create selections across multiple relations. This capability is commonly know as a 'Join' in the SQL language. Basically, this means that you can concatenate a record from one relation with a record from another relation. In order to know which record is concatenated with another record, there is a field in each relation that is used as a compare field. In the next tutorial, we will create a second relation for customer purchases. One of the fields in this new relation will be a "Customer_ID" field. For each new purchase, the customer's ID will be included with the purchase record. By using the 'Merge' command, you can create a selection of customers based on a particular item of purchase.

Tutorial 12

Merge

Before You Begin

This tutorial explains the 'Merge' command, which provides the ability to create selections across multiple relations. This capability is commonly know as a 'Join' in the SQL language. You need to understand all of the basic concepts of Jovis, especially the commands for multiple selections. The 'Merge' command is for advanced users. We also assume that the "Error" and "Warning" function handlers have been installed. Do not attempt to run any of the scripts in this tutorial without these handlers. You should have both the "DASCO Database" and the "DASCO Purchases.txt." available.

Basically, "merging" means that you can concatenate a record from one relation with a record from another relation. In order to know which record is concatenated with another, there is a field in each relation that is used as a compare field. In the this tutorial, we will create a second relation called "Purchases" which will contain each purchase a customer makes. One of the fields in this new relation will be a "Customer_ID" field, which duplicates the customer ID that is in our "Customers" relation. These two fields serve as our compare fields. For each new purchase, the customer's ID will be included with a newly created purchase record. By using the 'Merge' command, you can first create a selection of customers, and then merge it with their particular purchases.

Overview

Creating a "Merged Selection" is a two step process. The first step is to create a selection using the 'SetSelection' command. The second step uses the 'Merge' command to create a "Merged Selection". The selection created in the first step is not altered or deleted.

The 'Merge' command takes each row from the selection that was created in the first step, and, using a compare field in the selection, searches another relation or another selection for that same compare field's value. The value can be a number, such as an ID or date, or up to the first 32 characters of a text field. Whatever the value type, it must equal the value it is searching for if the records are to be concatenated.

This two step process makes it possible to create merged selections with great ease and tremendous flexibility.

Getting Started

To get started, you will need a minimum of two relations. (The Merge command does allow you to merge within a single relation; however, this situation is not common.) As was mentioned above, you need to create a compare field that duplicates information in both relations. This can be an ID number, a string of up to 32 characters, or a date. Once you have two relations with the specified compare fields, and the necessary data has been installed, you are ready to create a merged selection. For this tutorial we need to create a new relation called "Purchases", as well as the required fields, and three indexes. Here are the scripts for creating this relation in our "DASCO" database file:

```
on CreatePurchasesRelation
   get Jovis("CreateRelation","myDB","Purchases")
   get CreateJovisFields("Purchases", getPurchasesFldList("Fields"),
                     getPurchasesFldList("Types"))
   get CreateJovisIndexes("Purchases", getPurchasesFldList("IndexList"),
                     getPurchasesFldList("Sizes"))
end CreatePurchasesRelation

function getPurchasesFldList listType
   if listType = "Fields" then
      return "Customer_ID,Purch_Date,Stock_Item_Ref,Quant,"&
               Item_Descrip,Purchase_Amount"
   else if listType = "Types" then
      return "number,date,text,number,text,number"
   else if listType = "Indexes" then
      return "Customer_ID,Purch_Date,Stock_Item_Ref"
   else if listType = "Sizes" then
      return "10,4,6"
   end if
end getPurchasesFldList

function CreateJovisIndexes RelationName,IndexList,Sizes
   repeat with x = 1 to number of items of IndexList
      get Jovis("CreateIndex","myDB",RelationName,
                              item x of IndexList, item x of Sizes)
   end repeat
end CreateJovisIndexes

function CreateJovisFields RelationName,FieldList,TypesList
   repeat with x = 1 to number of items of fieldList
      get Jovis("CreateField","myDB",RelationName,
                     item x of FieldList, item x of TypesList)
   end repeat
end CreateJovisFields
```

Now we need to import some data (i.e. purchases) for our new relation. A text file called "DASCO Purchases.txt" has been provided for this purpose. Here is the script for importing this data:

```
on ImportToJovis
   put getFieldList("Fields") into fldList
   get Jovis("ImportData","myDB","Purchases",fldList,tab,return)
end ImportToJovis
```

There should now be 20 records in the "Purchases" relation.

To begin the merge process, use the 'SetSelection' command to create a selection which contains data that you want merged with additional information from another relation or another selection. Be sure to include the compare field in your list of selection fields (parameter four of the 'SetSelection' command). If you plan on merging with another selection, you will need to reserve your selection with the 'ReserveSelection' command. In our example, we will merge with the "Purchases" relation, rather than with a selection of purchases. Here is the first step, creating the initial selection:

```
   on CreateSelection
      put "Field Account_Start ≥ [1/1/95]" into aCriteria
      put "First_Name,Last_Name,Customer_ID" into fldList
      get Jovis("SetSelection","myDB","Customers",fldList,aCriteria)
      put Jovis("SelectionToVar","myDB","Customers",comma,return)
                    into cd fld "Display"
   end CreateSelection
```

The card field "Display" should now contain this information:

```
Mark,Jackson,3513
William,Anderson,8705
Susan,Abbado,7991
Doris,Epstein,4761
Barbara,Taylor,9553
Ruth,Addisen,2261
Dorthy,Quinn,2425
Mike,Stevens,2764
Tracy,Lippman,4729
Charles,Patrick,3876
```

Now you must script the parameters for the Merge command. There are six required parameters and a seventh one that is optional. Be sure to use the correct relation name for parameters two, four, and six. They control where a selection is located; parameter six in particular indicates where the "merged selection" will be located.

Equally important is that you make sure the selection field lists are correct. In parameter 3, you must only use fields which are used in the selection created in the first step. If you are using a named selection in parameter 4, then the same holds true for parameter 5; only those fields used in the named selection can be used. (Mistakes in these steps are the source of many "Invalid selection field" error messages.) The list of fields can be in any order, and you do not need to use all of the fields which are available in the named selection or the selection create in the first step.

Now we will concatenate the first and last names of those customers whose account was started on or after 1/1/95 with a description of their purchase item and their customer ID. Here is the complete script for creating this merged selection:

```
on CreateMergedSelection
   -- delete previously named selection, if it exists
   if "MERGED_TABLE" is in Jovis("ListSelections","myDB","Customers") then
      get Jovis("delSelection","myDB","Customers","MERGED_TABLE")
   end if
   --
   -- The first step is to create a selection
   put "Field Account_Start ≥ [1/1/95]" into aCriteria
   put "First_Name,Last_Name,Customer_ID" into fldList
   get Jovis("SetSelection","myDB","Customers",fldList,aCriteria)
   --
   -- Call the 'Merge' command
   get Jovis( "Merge",
   "myDB,Customers,Current,Customer_ID",
   "First_Name,Last_Name",
   "myDB,Purchases,UseRelation,Customer_ID",
   "Item_Descrip,Customer_ID",
   "myDB,Customers,MERGED_TABLE",
   "Exclusive" )
   --
   -- Display the "merged selection"
   put "First_Name,Last_Name,◊,Item_Descrip,Customer_ID" into fldList
   put Jovis("SelectionToVar","myDB","Customers",comma,
         return,fldList,"","MERGED_TABLE") into cd fld "Display"
end CreateMergedSelection
```

You should now see the following in card field "Display".

```
Mark,Jackson,Winter Coat,3513
William,Anderson,Desk Lamp,8705
Susan,Abbado,Dining Room Set,7991
Doris,Epstein,Silver Cake Platter,4761
Barbara,Taylor,Down Comforter,9553
Ruth,Addisen,Vacuum Cleaner,2261
Dorthy,Quinn,Snow Shoes,2425
Mike,Stevens,Waterman Fountain Pen,2764
Tracy,Lippman,Sandals,4729
Charles,Patrick,China Place Settings,3876
```

We now have a list of the customers concatenated with their purchases and Customer IDs.

The parameters for the 'Merge' command

We will now discuss each parameter of the 'Merge' command. You must pay particular attention to each item within the parameters in order to achieve the desired results. The 'Merge' command needs a great deal of information; therefore the parameters are not organized in the same style as the other Jovis commands. Being aware of this fundamental difference in parameter style can save you many hours of debugging time.

Param 2

We begin with parameter 2 as it is used in line 14 of our example above. This parameter provides the information about the selection created in the first step of the merge procedure. It requires four items separated by commas. The four items are:

1. Global File Identifier,
2. Relation Name,
3. Selection Name or "Current",
4. Compare Field Name

Item 1, the "Global file identifier," is the identifier for the database containing the relation we want to work with. Item 2 is the "Relation Name" for the relation where the selection is located. Item 3 is the name of the selection, if it is a reserved selection. If it is not reserved, you can use the literal "Current". (In line 14 of our example above, we use this "Current" selection for the "Customers" relation.) Item 4 is the name of the compare field. It MUST be a field in the selection.

This is the information used to search for matching records. There needs to be duplicate information in another relation so that an 'equals' operations can be performed between the selection and a given relation. All four items of this parameter must be present and correct for the merge to succeed.

Param 3

This parameter is an item-delimited list of fields which are available from the selection indicated in parameter 2. You can provide all of the fields in the selection or just a few, and they can be in any order. At least one field is required. In line 15 of our example, we have listed the first and last name fields. As required, these are fields included in the selection itself. Notice that the selection created in the first step has three fields - First name, Last name, and Customer ID - and that we are NOT including the Customer ID field. This is perfectly acceptable, even though the Customer ID field is going to be used as the compare field.

Param 4

Parameter 4 is the relation that we search using the compare field data in the selection we created in first step of this process. In line 16 of our example, which is the fourth parameter, we provide the information for the "Purchases" relation. This parameter consists of 5 items. Item 1 is the global file identifier for the file that contains the relation. Item 2 is the name of the relation that contains the other records being concatenated.

Item 3 can be either the literal "UseRelation", which indicates that the relation should be searched, or the literal "Current," which indicates that you want to use the current selection located at the relation indicated in item 2 for searching. You can also provide a named selection instead of one of the literals. If you use a named selection, be sure it's located in the relation listed in item 2 of this parameter.

Item 4 of this parameter is the name of the compare field. No matter which literal you use (either "UseRelation" or "Current", or a named selection), the compare field must be present. If you are using a selection, one of the fields in the selection must be the compare field. If you are using the literal "UseRelation", the relation must have the compare field you are planning on using.

If you use a relation (i.e. the "UseRelation" literal), the 'Merge' command will use the relation's index for performing the compare. This is straightforward and may require a little more time because of a few extra disk accesses. However, it is very important to understand that if the compare field is not indexed, every record in the relation will be accessed, and the requested fields for every record will be read into memory. This could easily cause an "out of memory" error if the relation has more records than memory will permit. For relations that have a few hundred records, there should not be reason for concern. When there are several thousand records in the relation, this compare field MUST be indexed.

Item 5 of this parameter is optional, and is used when the compare field is a text data type. By default, if this item is empty, Jovis uses the first 32 characters of the compare field for its searches. You can use as few a 1 or 2 characters if your data is unique enough to guarantee the correct record to record concatenation.

Param 5

This parameter consists of an item-delimited list of field names that you want concatenated to each record. These fields must be included in the relation that is being searched, or the selection indicated in item 3 of parameter 4. You can provide all, or just a few, of the fields in the relation or selection, but at least one is required. They can be listed in any order. Note that the compare field does not have to be included.

Param 6

When the 'Merge' command begins its processing, it requires a location and name for the merged selection. You need to provide three items of information for this parameter. Item 1 is a global file identifier. Item 2 is the name of a relation where the merged selection will be located. Item 3 is the actual name you want to assign to the merged selection. This name must be a unique name within the relation it is being assigned to. In our example above, the merged selection is assigned to the "Customers" relation, and is called "Merged_Table". We could have just as easily assigned it to the "Purchases" relation. Jovis allows you to assign it to any relation that is most convenient for you. The 'Merge' command will return an error if any of the three items of this parameter are not provided or incorrect.

Param 7

Parameter 7 indicates whether a merge is "Inclusive" or "Exclusive." An inclusive merge includes all of the records used during the compare, even if they do not match. The exclusive type accepts only those records whose compare fields match. This parameter is optional, and can be omitted. The default is "Exclusive".

Param 8

If the merge operation takes more than 2 seconds, a progress dialog box with a grow bar will appear. This parameter is for the message displayed with this dialog box. (To suppress the dialog's appearance, set the 'SetProperty' command's "delay" property to zero.) This parameter is optional; the default message is: "Merging data using table: [relation name]".

Using the "selection" commands with merged selections

Now that we have covered each of the parameters of the 'Merge' command, we can briefly discuss accessing a merged selection with the other "selection" commands, such as 'SelectionToVar' and 'GetSelectionRecord'. Because a merged selection is considered unique, these commands require additional information in order to access the selection's fields Note that the

'AppendSelection' and 'TrimSelection' commands cannot be used with merged selections. Also note that the commands 'CountSelection', 'DelSelection', 'DupSelection', and 'ReserveSelection' work exactly the same, regardless of the selection type.

For those commands that require additional information, a special field list delimiter is introduced. This delimiter is a diamond character "◊" (type shift/option - "v"), and indicates whether you want information from the fields, provided with the selection created in the first step, and/or information from the second part of the merged process.

By using the diamond character as a delimiter, you can indicate from which set of fields of the merged selection you want to retrieve information. For example, if we use the 'GetSelectionField' command to get the purchase description from the first row of the merged selection it would look like this:

```
   put Jovis("GetSelectionField", "myDB", "Customers", "◊Item_Descrip", 1,
                  "Merged_Table") into cd fld "Display"
```

The card field "Display" now contains "Winter Coat".

On the other hand, if we wanted to retrieve the last name from the first row of the merged selection we would script the field information in this matter:

```
   put Jovis("GetSelectionField","myDB","Customers",
               "Item_Descrip",1,"Merged_Table") into cd fld "Display"
```

The card field "Display" should now contain "Jackson".

The following provides examples and information for the selection commands that use the diamond delimiter as we demonstrated above.

ListSelectionFields

For merged selections, this command returns the names of all of the fields in the merged selection. A "◊" character will delimit those fields from the selection created in the first step, from the other fields. For example:

```
   put Jovis("ListSelectionFields","myDB","Customers",
                  "MERGED_TABLE") into cd fld "Display"
```

The card field "Display" should now contain:

```
      First_Name
      Last_Name◊Item_Descrip
      Customer_ID
```

ExportSelection

This command uses the same type of field list as the 'SelectionToVar' command does. Any of the field information you need from the first step of the merge process should come before the diamond delimiter ("◊" - type shift/option "v".). Always remember that this character is a separate item in the fields list.

```
   on mouseUp
      put "•" into fldDelim
      put return into recDelim
      put empty into filePath
      put "Set name and location for export file:" into SFPrompt
      put "First_Name,Last_Name,◊,Item_Descrip,Customer_ID" into fldList
      put empty into RecRange
      put "Exporting selection...Please standby" into ThermPrompt
      put "MERGED_TABLE" into NamedSelection
      get Jovis("ExportSelection","myDB","Customers",fldDelim,recDelim,
               filePath,SFPrompt,fldList,RecRange,ThermPrompt,NamedSelection)
   end mouseUp
```

GetSelectionCriteria

Merged selections are not created with criteria in the same way as standard selections are. The compare field always uses the equals operator. If you try to use this command on a merged selection, it will return empty.

GetSelectionField

Here are two more examples in addition to those above using the 'GetSelectionField' command:

```
   put Jovis("GetSelectionField","myDB","Customers","First_Name","6",
               "MERGED_TABLE") into cd fld "Display"

Returns: "Ruth"

   put Jovis("GetSelectionField","myDB","Customers","◊,Customer_ID","6",
               "MERGED_TABLE") after cd fld "Display"

Returns: "2261"
```

GetSelectionRecord

```
   Jovis("GetSelectionRecord","myDB","Customers","8",
               "false","MERGED_TABLE") into cd fld "Display"

   Jovis("GetSelectionRecord","myDB","Customers","◊8",
               "false", "MERGED_TABLE") into cd fld "Display"
```

GetSelectionStats

```
   put Jovis("GetSelectionStats","myDB","Customers","◊,Customer_ID",
               "MERGED_TABLE") after cd fld "Display"
```

RestoreSelection

This commands works exactly the same for either standard or merged selections. The main reason for making a merged selection the "Current" or "Active" selection is so that you can use the 'SortSelection' command. It is not possible to sort named selections. (See 'SortSelection' below.)

SelectionToVar

In our example above, the completed merge creates a newly named selection called "MERGED_TABLE". Because the merged selection is a named selection, you can use the eighth parameter of the 'SelectionToVar' command to display the selection. The sixth parameter of this command, the selection fields list, is constructed using the "◊" character (shift/option - 'v') which delimits the fields of the merged selection. Note that the "◊" character is a separate item in the fields list. Our example above shows how to do this. Here it is again:

```
         put "First_Name,Last_Name,◊,Item_Descrip,Customer_ID" into fldList
         put Jovis("SelectionToVar","myDB","Customers",comma,
            return,fldList,"","MERGED_TABLE") into cd fld "Display"
```

SortSelection

Because the 'SortSelection' command requires that the selection be a "Current" or "Active" selection, you must first make the merged selection a "Current" selection using the 'RestoreSelection' command. Notice once again that we the use of the diamond character in the list of fields, and that it is a separate item. In order to show how this command works, we provide two complete handlers.

```
   on mouseUp
      get Jovis("RestoreSelection","myDB","Customers","Merged_Table")
      get Jovis("SortSelection","myDB","Customers","◊,Customer_ID,d")
      put "First_Name,Last_Name,◊,Item_Descrip,Customer_ID" into fldList
      put Jovis("SelectionToVar","myDB","Customers",
                  comma,return,fldList) into cd fld "Display"
   end mouseUp
```

The card field "Display" should now contain:

```
Barbara,Taylor,Down Comforter,9553
William,Anderson,Desk Lamp,8705
Susan,Abbado,Dining Room Set,7991
Doris,Epstein,Silver Cake Platter,4761
Tracy,Lippman,Sandals,4729
Charles,Patrick,China Place Settings,3876
Mark,Jackson,Winter Coat,3513
Mike,Stevens,Waterman Fountain Pen,2764
Dorthy,Quinn,Snow Shoes,2425
Ruth,Addisen,Vacuum Cleaner,2261
```

```
   on mouseUp
      get Jovis("RestoreSelection","myDB","Customers","Merged_Table")
      get Jovis("SortSelection","myDB","Customers","Last_Name")
      put "First_Name,Last_Name,◊,Item_Descrip,Customer_ID" into fldList
      put Jovis("SelectionToVar","myDB","Customers",
                     comma,return,fldList) into cd fld "Display"
   end mouseUp
```

The card field "Display" should now contain:

```
Susan,Abbado,Dining Room Set,7991
Ruth,Addisen,Vacuum Cleaner,2261
William,Anderson,Desk Lamp,8705
Doris,Epstein,Silver Cake Platter,4761
Mark,Jackson,Winter Coat,3513
Tracy,Lippman,Sandals,4729
Charles,Patrick,China Place Settings,3876
Dorthy,Quinn,Snow Shoes,2425
Mike,Stevens,Waterman Fountain Pen,2764
Barbara,Taylor,Down Comforter,9553
```

Script Examples

The following script provides a number of examples that use the selection commands as well as our 'Merge' command. In the first example above, we merged a selection of customers with the "Purchases" relation which resulted in a list of customers and a description of the items they purchased. In the following example, we reverse the type of merge; it merges the purchases against a selection of customers. This example is more memory intensive because we use a named selection called "Selected_Purchases" to searching within.

```
on mouseUp
   global JovisErrorCode
   set cursor to watch
   --
   -- clear out any former selections by the given names
   if "Illinois_Customers" is in Jovis("ListSelections","myDB","Customers") then
      get Jovis("delSelection","myDB","Customers","Illinois_Customers")
   end if
   if "Selected_Purchases" is in Jovis("ListSelections","myDB","Purchases") then
      get Jovis("delSelection","myDB","Purchases","Selected_Purchases")
   end if
   if "MERGED_TABLE" is in Jovis("ListSelections","myDB","Customers") then
      get Jovis("delSelection","myDB","Customers","MERGED_TABLE")
   end if
   --
   put "Purchase_Amount,Purch_Date,Customer_ID" into fldList
   put "Field !RecID > [0]" into cri
   get Jovis("SetSelection","myDB","Purchases",fldList,cri)
   --
   -- let's see what we've got via the script debugger:
   put Jovis("SelectionToVar","myDB","Purchases","•",return,"*") into bin
   --
   -- Reserve the 'Purchases' current selection
   get Jovis("ReserveSelection","myDB","Purchases","Selected_Purchases")
   --
   put "First_Name,Last_Name,State,Zip,Customer_ID" into fldList
   put "FIELD State = [Il]" into cri
   get Jovis("SetSelection","myDB","Customers",fldList,cri)
   --
   -- let's see what we've got via the script debugger:
   put Jovis("SelectionToVar","myDB","Customers","•",return,"*") into bin
   --
   -- Reserve the current selection
   get Jovis("ReserveSelection","myDB","Customers","Illinois_Customers")
   --
   put Jovis("ListSelections","myDB","Purchases") into Purchases
   put  Jovis("ListSelections","myDB","Customers") into Customers
   --
   -- MERGE:
   get Jovis("Merge",
   "myDB,Purchases,Selected_Purchases,Customer_ID",
"Customer_ID,Purchase_Amount,Purch_Date",
"myDB,Customers,Illinois_Customers,Customer_ID",
"First_Name,Last_Name,Zip",
"myDB,Customers,MERGED_TABLE",
"Exclusive")
--
-- Make merged selection the current selection in order to sort it:
get Jovis("RestoreSelection","myDB","Customers","MERGED_TABLE")

-- Sort merged selection:
get Jovis("SortSelection","myDB","Customers","◊,Last_Name")

-- Now let's display it:
put "Customer_ID,Purchase_Amount,Purch_Date,◊,First_Name," &
         "Last_Name,Zip" into fldList
put Jovis("SelectionToVar","myDB","Customers",comma,return,fldList)
         into cd fld "Display"

-- For the fun of it, revert the merged selection
-- back to a reserved selection.
get Jovis("ReserveSelection","myDB","Customers","MERGED_TABLE")
--
-- Get the merged selection's field list
put return & return & Jovis("ListSelectionFields","myDB","Customers",
         "MERGED_TABLE") after cd fld "Display"
--
put return & return & Jovis("GetSelectionField","myDB","Customers",
         "Purch_Date","3","MERGED_TABLE") after cd fld "Display"
put comma & Jovis("GetSelectionField","myDB","Customers",
         "◊,Last_Name", "3","MERGED_TABLE") after cd fld "Display"
--
put return & return & Jovis("GetSelectionStats",
         "myDB","Customers","◊,Zip","MERGED_TABLE") after cd fld "Display"
put return & return & Jovis("GetSelectionStats", ¬
         "myDB","Customers","Purch_Date","MERGED_TABLE") ¬
            after cd fld "Display"
--
-- get the actual record for the third row of
-- the selection created by the first step
-- note that 'GetSelectionRecord' takes a sixth parameter
-- for the named selection
put Jovis("GetSelectionRecord","myDB","Customers","3",
         "False","MERGED_TABLE")  into SelectedCustomerRecNum_3
--
-- get the actual record for the third row of the merged selection
-- note the '◊' character in the forth parameter, (shift/option 'v')
put Jovis("GetSelectionRecord","myDB","Customers","◊3",
         "False", "MERGED_TABLE")  into SelectedPurchasesRecNum_3
--
-- what named selections does the "Purchases" relation have?
put Jovis("ListSelections","myDB","Purchases") into Purchases
--
```

```
   -- what named selections does the "Customers" relation have?
   put  Jovis("ListSelections","myDB","Customers") into Customers
   --
   -- Clear named selections from memory:
   get Jovis("delSelection","myDB","Customers","Illinois_Customers")
   get Jovis("delSelection","myDB","Purchases","Selected_Purchases")
   get Jovis("delSelection","myDB","Customers","MERGED_TABLE")
end mouseUp
```

Error Messages That You May Encounter

Warning:
No records selected

Errors:
Invalid selection field - One of the fields in one of the Merge command's parameters is not valid. You need to carefully go over each field name for spelling and typos, and that it is indeed included in selection(s) and/or the relation you are merging with.

Global name missing or invalid
Invalid character for merged selection name
Invalid Selection field name
Selection: [Name] not found, or it is empty
Invalid selection row

Script Errors
"Not a valid Jovis command" - Either a typo or the first parameter is not in quotes correctly.

"Never heard of that function name" - misplaced quotes.

What You Should Have Learned

The 'Merge' command is certainly the most involved command that Jovis provides. While the number of details may perhaps seem overwhelming at first, once you have used it a few times, it will become simple to work with. You should have learned that the parameters for this command are styled differently than the other Jovis commands. If any of the items in each of the parameters is incorrect, the command will fail.

What to learn Next

This tutorial marks the last of the relational tutorials. If you have worked your way from the first tutorial to this one, you deserve a round of applause! We hope that you have not only acquired the basic skills to use Jovis for you needs, but have gained an insight into how Jovis works.

There are four more tutorials which discuss the "Architectural" commands and how to use them within a relational database. We strongly recommend that you continue learning Jovis with these tutorials, particularly if you intend to use the multimedia capabilities.