UniVerse Server Developer Self Paced Training

Brian Leach

Setting the Stage

UniVerse is, first and foremost, an application platform.

Although UniVerse is commonly described as a database, that term can be misleading when you compare it alongside other mainstream database products. If you have previously completed the first volume in this series, *UniVerse Enquiry Self Paced Training*, you will already be familiar with the fact that UniVerse offers much more than data storage and retrieval and that the approach to building and running applications - especially in terms of where the critical business logic should live - is very different than for other platforms.

Modern applications are multi-layered, messy and sprawling things – here is a database, there is Java business layer, on top of these sit .NET forms and websites using a whole panoply of different technologies. Surrounding those are the mechanisms for automated testing, source control, continuous deployment and all the tools necessary for managing such an arrangement and keeping those layers talking together and in sync.

At the heart of the UniVerse platform is an application engine. This, along side the storage model and enquiry languages, allows you to write complete business processes that sit within the database itself. From a development, testing and deployment point of view, this makes life very much simpler – and from a performance standpoint it removes the necessity to drag large amounts of data into a client or middle tier only to manipulate it and dump it all back again. The result is simpler, more cost effective, and generally quicker transactional applications. Put simply, the closer the logic is to the data, the faster and cheaper things become.

There are other advantages to this approach. By centralizing all the working logic of the appplicaton inside the database platform itself, it ensures that it is there at the lowest tier ready to be accessed by whatever choice of fronting technology you may adopt. So the same processes will work in the same way, regardless of whether they are called from Java, from .NET, from RESTful services or any of a dozen other technologies. Which means in turn that there is less likelihood of code duplication and the management nightmares of keeping multiple code sets updated with business changes.

Historically, UniVerse applications were entirely self-contained and originally text based, running inside the database server and accessed using terminal emulators. In fact, you would be amazed how many such legacy applications still exist, running critical systems for finance, trading, manufacturing and other industries where performance and stability are the key factors.

Today, it is unlikely that you will be writing systems that live entirely in the UniVerse space. More likely, you will be writing the functional parts of an application – the back end rules, validations and business processes that do the real work behind the scenes – to be called from whatever other front end languages and frameworks your organization uses; and stand-alone programs that provide administrative functions and background processing.

UniVerse supports the concept of 'phantom' processes – sessions that run in the background to perform specific tasks – and these can be used to offload long running operations or scheduled tasks such as trade submissions and pricing, order fulfilment or exports to remote data sources. With thoughtful design they can also be used to optimize the user experience: an application backing website might choose to generate and cache the results of the most frequent requests in advance to reduce waiting times and contention.

At time of writing, the tools of choice for middleware developers are the Microsoft .Net platform, Java and the weath of AJAX-enabled web front ends such as node.js. Essentially when it comes to writing front end code for UniVerse most developers use exactly the same technology stacks that you will find used everywhere else: my desktop normally has mvDeveloper (UniVerse) open on one monitor and Visual Studio on the other.

The software world rarely stands still, and the skills you learn today can lead you into practically every technology tomorrow if you are curous and minded to do so. Over the years I have developed solutions and taught new students using Visual Basic, ASP, PHP, Delphi, Java, VB.Net, ASP.NET, C#, Silverlight, a bunch of different AJAX frameworks and a host of other technologies: the common thread being that all of these have been backed by UniVerse. That's what keeps the life of a UniVerse developer interesting.

Using @RECORD.

A very powerful use of this function takes place when you apply it to a special system variable named `@RECORD` , one of the system variables we delayed looking into in the section on system variables above.

Whenever the enquiry processor reads a record from the file it places a copy of the record body into the `@RECORD` system variable. This is an exact replica of the record as it is held in the file, complete with field, value and subvalue marks, and it allows you to get at any part of the record regardless of whether or not there is a data descriptor defined for it. The processor also places a copy of the record key into a system variable called `@ID`, which is a little confusing since every UniVerse file is initially created with a single key-defining dictionary item, also called @ID.

Do This:

To see the content of the current record in a format you can easily read, the `CONVERT()` function can be used to replace the system delimiters with characters that can be displayed correctly. By convention we use the following characters to denote a field, value and subvalue mark respectively:

^	Field mark
]	Value mark
\	Subvalue mark

By using the `CONVERT()` function with `@RECORD` you can create a field for BOOK_SALES called RAW_RECORD to show the actual structure, with the following expression:

```
CONVERT(@FM:@VM:@SVM,"^]\", @RECORD)
```

Note

The Editor uses the caret (^) as a special escape character. If you are using the UniVerse editor, you need to type two carets together ("^^") to insert a single caret ("^")

When you list this, you can see the underlying structure of each of the BOOK_SALES records:

```
LIST BOOK_SALES RAW_RECORD

BOOK_SALES. 13660*37800*1
Raw Record. MOFFET^PETER^31 Coppice Lane]HALESOWEN]WEST MIDLANDS]
            B 3YB^201]110]4]96]106]194^1]1]1]1]1]1]1]1^2499]645
            7]1300]1499]1595]1799]1299]1390]0]0]0]0]0

BOOK_SALES. 13512*63000*1
Raw Record. MANSER^BOB^31 Little Lane]CHEPSTOWE]GWENT]MP6 1EU^95]
            64]207]74]141]1^1]1]1]1]1]1]1]1]1^999]1699]799]599]12
            99]1499]1099]999]1599^0]0]0]0]

BOOK_SALES. 13589*43200*1
Raw Record. BUCKERIDGE^SHIRLEY^48 Willingham Road]DALKEITH]MID LO
            THIAN]EH22 7VM^57]127^1]1]1]1]1^1300]1499]899]1300]24
            99^0]0]0]0]0
```

Any part of the underlying record can be accessed by using the `EXTRACT()` function with the `@RECORD` system variable. This means that the ADDR1 field that you just added to the BOOK_SALES can be rewritten as below, using the `EXTRACT()` function to return the first value of the third (address) field.

Field	DICT BOOK_SALES ADDR1
1	I First line of address
2	EXTRACT(@RECORD,3,1)
3	
4	Address Line 1
5	20L
6	S

Using `EXTRACT()` is neater than using the equivalent delimited substring extraction, especially since there is also a shorthand notation that can be used to the same effect. Fields, values and subvalues from an expression can be accessed directly by using a special operator:

```
Expression<field>
Expression<field,value>
Expression<field,value,subvalue>
```

This allows the expression above to be rewritten once again, but this time in a much shorter format:

Do This:

Field	DICT BOOK_SALES ADDR1
1	I First line of address
2	@RECORD<3,1>
3	
4	Address Line 1
5	20L
6	S

This operator is truly a shorthand - internally this compiles into an `EXTRACT()` function – but one that is much more intelligible as well as being easier to type.

Lab 3.8

Rewrite the POSTCODE field in the BOOK_SALES to use the extraction operators.

Paragraphs

When you have several sentences, they make up a paragraph – and thus the Paragraph is the name given to the simplest batch structure in a UniVerse system.

A paragraph contains a number of commands that are normally executed in strict order. Each paragraph can contain several lines with the following format:

Field 1	PA description
Field 2	command
Field 3+	commands

The first field begins with the letters 'PA' and, as you can probably guess by now, this can be optionally followed by a description. The second and following fields onwards hold the commands to run in sequence with each command starting on a new line. As with sentences, long or complex commands can be broken over multiple lines using the continuation character (_).

Paragraphs are used to run multiple commands in sequence, and can include in-line prompts as below:

Do this:

Field	VOC CUSTOMER_SALES_PA
1	PA Book sales customer search
2	* Perform a search for sales orders for a customer
3	CS
4	DISPLAY Customer Orders Search:
5	SORT BOOK_SALES WITH SURNAME LIKE "…<<Surname>>…" _
6	OR WITH SURNAME SAID "<<Surname>>" _
7	HEADING "Customer search for <<Surname>>"
8	DISPLAY That's all folks

Paragraphs can include comment lines that begin with an asterisk. There must be a space between the asterisk and the start of the comment text or the entry will be interpreted as a call to a special type of global program that you will meet later on.

To display messages from within a paragraph you can use the DISPLAY command. This is directly followed the text to be displayed, which does not need to be enclosed in quotation marks.

Passing Command Line Parameters

If you are going to run a parameterized command as part of a wider batch of commands, you do not necessarily want the paragraph to stop half way through and ask for details – especially if it means waiting for commands that are going to take a long time to run! Even if you do not mind running interactively, it can be neater to be able to pass a prompted value to a paragraph on the command line, rather than as the response to a prompt on the screen.

An inline prompt is made up of several elements, of which the prompt text is only one part. You can add codes to the prompt that will affect the way in which the prompt is requested, and validation to ensure that only sensible values can be accepted.

The action codes appear before the prompt text, separated from the text and from each other by means of commas. The most commonly used action codes are:

@(CLR)	Clear the screen.
@(c,r)	Place prompt at column c, row r of the screen.
A	Always prompt.
Cn	Take element n from the command line.
F(file)id,fm	take the answer from a field in a record.
In	take element n from the command line or prompt.

The C and I codes and the A code can only be used in the context of a paragraph. The others can be used in paragraphs or in sentences.

Cursor Positioning.

The @(c,r) action positions the cursor at a specified location on screen to ask for a value. This can be used when presenting multiple prompts on screen, though frankly you are better to use a PROC or UniVerse Basic program if you want to control the screen layout.

Where cursor positioning is required, the @(c,r) action can be added before the prompt text, and positions the cursor using coordinates relative to the top left hand corner of the screen. The top left corner of the screen is @(0,0) whilst the bottom right is probably (79,24) depending on your terminal settings.

Do this:

You can extend the customer sales paragraph to include some simple cursor positioning:

Field	VOC CUSTOMER_SALES_PA
1	PA Book sales customer search
2	* Perform a search for sales orders for a customer
3	CS
4	DISPLAY Customer Orders Search:
5	SORT BOOK_SALES WITH SURNAME LIKE "…<<@(10,5),Surname>>…" _
6	OR WITH SURNAME SAID "<<Surname>>" _
7	HEADING "Customer search for <<Surname>>"
8	DISPLAY That's all folks

Command Parameters.

Of more practical benefit are the `C` and `I` action codes. These both allow you to supply the value for a prompt as a parameter on the command line. The `C` and `I` codes are both followed by a number which gives the element on the command line to be used. The name of the paragraph itself counts as the first element, so if you wanted to call the CUSTOMER_SALES_PA paragraph passing the surname:

```
CUSTOMER_SALES_PA SMITH
```

the inline prompt for the Surname would need to specify 2 for the `C` and `I` codes.

Where the C and I codes differ from one another is in the action they will take when the requested element is missing from the command line. The C (Command) code will only look at the command line, and so will substitute an empty string in place of the prompt. The I (Interactive) code will look first at the command line, and will then prompt interactively if the element is not found.

Do This:

Change the customer search to pass the surname on the command line or to prompt for the surname if it is not present:

Field	VOC CUSTOMER_SALES_PA
1	PA Book sales customer search
2	* Perform a search for sales orders for a customer
3	CS
4	DISPLAY Customer Orders Search:
5	SORT BOOK_SALES WITH SURNAME LIKE "…<<@(10,5),I2,Surname>>…" _
6	OR WITH SURNAME SAID "<<Surname>>" _
7	HEADING "Customer search for <<Surname>>"
8	DISPLAY That's all folks

Notice that the action code does not have to be repeated for the subsequent uses of the prompt.

The A action code is used to ensure that the prompt is repeated each time the command is run. Normally UniVerse will only prompt once for the prompt within a single paragraph.

Lab 5.7

Enhance your AUTHOR_SEARCH so that it will accept an author name passed from the command line. Note that if the author name includes spaces, the name must be passed surrounded by double quotes. These will not be substituted into the prompt.

Opening a UniVerse File

Before you can perform any read or write operations against a UniVerse file from within UniVerse Basic the file must be explicitly opened. This associates the file with a special type of variable known as a File Variable or File Pointer through which all subsequent actions against that file are performed.

A File Variable contains an internal structure giving details about the underlying file: the location of the file, the file type, modulus and separation and so forth. None of this information is directly visible – you cannot, for example, `CRT` a file variable without being presented with a sulky error message about the data type. The only things you can do with a file variable are legitimate file operations, or assignments to another file variable.

Files are opened using the `OPEN` statement:

```
OPEN filename TO filevariable {THEN|ELSE}
```

For example:

```
Open "VOC" To F.VOC Else STOP "Cannot open VOC"
```

If a dictionary is being opened, there are two possible syntaxes:

```
Open "DICT", "VOC" To D.VOC Else STOP "Cannot open VOC"
Open "DICT VOC" To D.VOC Else STOP "Cannot open VOC"
```

Tip

I use upper case to distinguish special variables, global variables and constants from regular local variables so that I can see immediately that there is something different about these when scanning quickly through code.

There is a standard convention that files variables are prefixed with an 'F.', such as F.ORDERS and F.CUSTOMERS, and dictionaries with ‘D.’ as in D.VOC.

Statement Branching

The OPEN statement is the first statement you have encountered in this course to offer branching.

When you tell UniVerse to open a file there are two possible outcomes: the file exists and can be opened successfully, or the file may not. You may have mistyped the name or you may not have operating system privileges on the file.

When UniVerse Basic encounters a statement that can potentially fail, UniVerse normally handles these situations by offering the developer two possible branches to the statement: a THEN branch and an ELSE branch. If the statement succeeds, the code will follow the THEN branch. If the statement fails, the ELSE branch is followed. It is a simple and neat form of error trapping that maintains the flow of the program and keeps the errors locally attached to the statements that have raised them.

The fundamental rule is that for any statement that supports branching, you can specify either a THEN clause, an ELSE clause or both. You **must** specify at least one of these branches or the compiler will reject your code.

There are two forms of syntax for the code that follows a particular branch. If the branch consists of only a single statement you can write this directly after the THEN or ELSE keyword on the same line:

```
Open "VOC" To F.VOC Else ABORT "Cannot open the VOC File"
```

The ABORT statement terminates a program printing an optional message. This is more powerful than the STOP statement and is reserved for fatal error conditions.

Single line branches can be applied to the THEN or ELSE clauses, or both:

```
Open "VOC" To F.VOC Then Crt "VOC opened" Else ABORT "Cannot
open the VOC"
```

Single line branches are fine for simple operations such as terminating a program. In most cases, you will need to perform a range of activities in each branch. That in turn means that UniVerse needs to know where each branch will end.

Multiple line branches consist of one or more statements starting on the line following the THEN or ELSE clause and terminating with an END statement, as below:

```
Open "VOC" To F.VOC Then
   Crt "VOC file opened"
End
```

or:

```
Open "VOC" to F.VOC Else
   Crt "Cannot open the VOC File"
   STOP
End
```

If you want to combine both THEN and ELSE clauses both branches terminate with an END. The END ELSE is usually placed together on the same line:

```
Open "VOC" To F.VOC Then
   Crt "Opened the VOC File"
End Else
   Crt "Could not open the VOC File"
   STOP
End
```

Remember you can use the STATUS() function to return more information on failures for many commands and functions.

Getting File Information

The FILEINFO() function returns a specific piece of information requested by passing a file variable into the function along with a number that determines what information should be returned.

```
Value = FILEINFO( filevariable, number)
```

The following table lists the FILEINFO() request values:

0	Valid file handle (true or false)
1	VOC name of the file
2	pathname of the file
3	File format: 1 hashed, 3 dynamic, 4 directory, 5 sequential, 7 distributed
4	Hashing algorithm used
5	Current modulus (dynamic file)
6	Minimum modulus (dynamic file)
7	Group size in 1k units
8	Large record size (dynamic file)
9	Merge load parameter (dynamic file)
10	Split load parameter (dynamic file)
11	Current load (dynamic file)
12	Node hosting file (remote file)
13	Secondary indices exist (true or false)
14	Current line number
15	Part number (distributed file)
16	Status (distributed file)
17	Recovery type
18	Recovery id
19	Fixed modulus (true or false)
20	Map used by NLS

Lab 3.1

Create a program called fileinfo. This will request the name of a UniVerse file and will then attempt to open the file.

If the open succeeds it will print out the file name, file path and file format on the terminal.

If the open does not succeed it will stop with a suitable error message.

Error Trapping

Now that we have introduced the concept of statement branching on success of failure, you need to be aware that some statements can support other types of branch. For the OPEN statement there is a third branch condition signalled by an ON ERROR clause.

The full specification of the OPEN statement is:

```
OPEN filename TO filevariable [ON ERROR clause]{THEN|ELSE
clause}
```

Normally you will only be worried about whether or not you can open the file, and so the most programs restrict themselves to using just the THEN and ELSE clauses. It is, however, possible to distinguish between errors caused by the logic of the program (such as getting the name of a file wrong) and fatal errors raised at runtime due to problems such as a bad file structure.

Fatal error conditions are handled by the ON ERROR clause, if present, or the ELSE clause if not. The ON ERROR clause follows the same branching pattern as the THEN and ELSE clauses and must precede these:

```
Open "VOC" To F.VOC On Error
   ABORT "A fatal error has occurred opening the VOC File - Status
is ": Status()
End Then
   Crt "Opened the VOC File"
End Else
   Crt "Could not open the VOC File"
   STOP
End
```

Notice the use of the STATUS() function to return an error code. If you recall from the discussion on the ICONV() function above, STATUS() is called to return information about the success of failure of a preceding operation. The STATUS() following an OPEN statement is set to the file type if the OPEN succeeds or on failure to one of the following negative values:

-1	The filename was not found in the VOC file.
-2	The filename or file is null, the file cannot be opened over uv/Net or other generic errors.
-3	You do not have permission to access a UniVerse file in a directory
-4	An access error appears when you do not have operating system permissions or if DATA.30 is missing for a type 30 file.
-5	The operating system detected a read error.
-6	The lock file header cannot be unlocked.
-7	Invalid file revision or wrong byte-ordering for the platform.
-8	Invalid part file information for a distributed file.
-9	Invalid type 30 file information exists in a distributed file.
-10	A problem occurred while the file was being rolled forward during warmstart recovery. Therefore, the file is marked "inconsistent."
-11	The file is a view; therefore it cannot be opened by a BASIC program.
-12	No SQL privileges exist to open the table.
-13	An index problem exists.
-14	The NFS file cannot be opened.

Do you need the On Error clause?

The original MVDBMS systems did not support an ON ERROR clause, and very few UniVerse programmers make use of it. You can certainly survive without it for regular database operations, but it is always a good idea to program as defensively as possible and to provide your support team (especially if that happens to also be yourself) with as much information as possible should an error occur.

You should certainly consider using ON ERROR clauses for new code, through you may wish to wrap this into an external function. In fact, you will be writing just such a function later in this course.

Controlling the XML Format

Using RetrieVe or SQL to generate XML output is quick and convenient, but does not provide a whole lot of control over the format of the XML that is produced. That is generally fine if you are both the producer and consumer of the XML document, or if you have the authority to dictate what the document format should be. Often though the format may have been defined externally, especially in the case of data passed between different parties or that needs to adhere to some industry or other standard representation.

For complex formats, and particularly for those involving significant levels of nesting, you may find that you need to resort to the programming APIs that we will be covering later in this chapter. Before automatically reaching for these, you may discover that you can handle some of the document formats by using the enquiry language in conjunction with a mapping file. This combination allows you to determine - to a limited extent - the format of the XML that will be produced and to use this either to create a fully formed document or to build a snippet that can be captured and inserted into a document template.

To define the XML format generated by a RetrieVe or SQL query you must first create a mapping file. This is a document in XML format that must reside in a directory named "&XML&" in the account in which the enquiry will be run. If your account does not already contain an &XML& file, you will need to create this as a type 19 (directory) file:

```
CREATE.FILE &XML& 1,1 1,1,19
```

The mapping file defines the way in which RetrieVe and SQL will treat particular fields and also the names given to the root, record and association elements in the generated document. The map can be attached to more than one enquiry statement.

Before we look at the structure of the mapping file, it is worth running a simple example to see the mapping in effect.

The following example shows a simple mapping record named 'titles_out.map' which defines the output format for a BOOK_TITLES XML listing:

```
<?xml version="1.0" encoding="utf-8" ?>
<U2XML-mapping xmlns:U2XML="http://www.ibm.com/U2-XML">
<U2XML:mapping record="TitleRec" root="TitlesInStock" schema="type" />
   <U2XML:mapping file="BOOK_TITLES" field="ISBN" treated-as="element" />
<U2XML:mapping file="BOOK_TITLES" field="@ID" treated-as="attribute"
      map-to="TitleID" />
   <U2XML:mapping file="BOOK_TITLES" field="SHORT_TITLE" map-to="Title"
      treated-as="element" />
   <U2XML:mapping file="BOOK_TITLES" field="AUTHOR_NAME" type="S"
      map-to="Author"
      treated-as="element" />
</U2XML-mapping>
```

This mapping file can be applied to an XML listing by adding the XMLMAPPING keyword followed by the name given to this record in the &XML& file.

Do this:

```
SORT BOOK_TITLES ISBN SHORT_TITLE AUTHOR_NAME TOXML XMLMAPPING
"titles_out.map"
```

```
<?xml version="1.0" encoding="UTF-8"?>
<TitlesInStock
  xmlns:U2XML="http://www.ibm.com/U2-XML">
<TitleRec TitleID = "1">
  <ISBN>0563525428</ISBN>
  <Title>Just William: No. 6 (BBC Radio Collection)</Title>
  <Author>Richmal Crompton</Author>
</TitleRec>
<TitleRec TitleID = "2">
  <ISBN>0955064007</ISBN>
  <Title>Just So Stories</Title>
  <Author>Rudyard Kipling</Author>
</TitleRec>
<TitleRec TitleID = "3">
  <ISBN>0001050478</ISBN>
  <Title>The Duchess of Malfi (abridged version)</Title>
  <Author>John Webster</Author>
</TitleRec>
<TitleRec TitleID = "4">
  <ISBN>1901768384</ISBN>
  <Title>Great Expectations</Title>
  <Author>Charles Dickens</Author>
</TitleRec>
<TitleRec TitleID = "5">
  <ISBN>9626343427</ISBN>
  <Title>The Importance of Being Earnest</Title>
  <Author>Oscar Wilde</Author>
</TitleRec>
```

If you compare this example to an XML listing of the BOOK_TITLES created without the mapping file you will notice a number of significant differences:

- The name of the record set is now TitlesInStock in place of the ROOT.
- The name of each title record is now TitleRec not BOOK_TITLES.
- The name of the @ID field is now TitleID not _ID
- The SHORT_TITLE and AUTHOR_NAME fields have been renamed.
- The TitleID has been changed into an attribute.

As you can see from that last change, the mapping file opens up the possibility of creating mixed-mode documents including both attributes and elements.

Notice however that the mapping file only controls how fields and other elements should be processed *if they appear* in the listing. It does not determine in any way what the content of the listing should be: you still get to decide at the point of building the enquiry which of those fields you want to include. The listing does not need to include all of the fields mentioned in the mapping file, and likewise any additional fields that are not in the mapping file will still be included, but will be shown using their default presentation:

```
SORT BOOK_TITLES ISBN SHORT_TITLE AUTHOR_NAME UNITS TOXML XMLMAPPING
"titles_out.map"

<TitleRec TitleID = "1" UNITS = "3">
  <ISBN>0563525428</ISBN>
  <Title>Just William: No. 6 (BBC Radio Collection)</Title>
  <Author>Richmal Crompton</Author>
</TitleRec>
```

This means that the mapping file is not specific to one listing. If you so wished, you might create a mapping file holding a sensible layout policy for each of the major files you are likely to want to query into XML to ensure that the data is reported consistently.

Building a Document

A JSON style document is an extensible, nested object capable of handling elaborate structures. As I wrote in the introduction to this section, there are really only two parts to this, which are used in combination: name-value pairs and lists. Building up a JSON document is a question of adding these elements.

To build this document in code you would begin by creating a new, empty UDO. For this you require the UDOCreate function passing it a UDO_OBJECT flag. The function will return a handle to an empty object that you can then extend:

```
Ok = UDOCreate(UDO_OBJECT, hJSON)
```

Do remember that you need to free this when you have finished with it.

Once you have your object available, you can extend it by adding named values, also known as properties. The UDOSetProperty function adds a new property to the object, or replaces the value for an existing property of the same name.

```
Ok = UDOSetProperty(hJSON, name, value)
```

For Example:

```
Ok = UDOSetProperty(hJSON, "titleId", "1")
Ok += UDOSetProperty(hJSON,"title", "Just William: No. 6")
Ok += UDOSetProperty(hJSON,"author","Richmal Crompton")

If Ok <> UDO_SUCCESS Then
   GoSub ShowError
End
```

Lab 6.1

Create a program udo1 that will request a book title and fill out a UDO with the following name pairs:

titleId
title
author
price
ISBN

Adding a Nested Object

JSON documents are tree like structures with nested elements. Adding a nested element in UDO is a two stage process: first you need to create the element and then you can add it to its parent.

Let's take the lab excercise you completed above. Instead of just returning the author name, you can extend it to return a simple nested object containing the author id and name together:

Do This:

```
AuthorId = TitleRec<BOOK_TITLES.AUTHOR_ID>
ReadV AuthorName From F.AUTHORS, AuthorId, BOOK_AUTHORS.FULLNAME Else
   AuthorName = 'Unknown'
End

Ok = UDOCreate(UDO_OBJECT, hAuthor)
Ok += UDOSetProperty(hAuthor,"id", AuthorId)
Ok += UDOSetProperty(hAuthor,"name", AuthorName)
Ok += UDOSetProperty(hJSON,"author",hAuthor)
```

This should now contain the author object as below:

```
Enter title id : ?1
{
        "titleId":      "1",
        "title":        "Just William: No. 6 (BBC Radio
Collection)",
        "price":        "15.99",
        "ISBN": "0563525428",
        "author":       {
                "id":   "1",
                "name": "Richmal Crompton"
        }
}
```

Adding an Array

An array or list is constructed in a similar way.

First you need to create a new element of type UDO_ARRAY using the UDOCreate function.This will return a handle to an empty array, which you can populate by adding new elements using the UDOArraySetItem or UDOArrayInsertItem functions:

Ok = UDOArraySetItem(hArray, index, value)
Ok = UDOArrayInsertItem(hArray, index, value)

The index is an element number starting from 1. If you specify a number that is greater than the array length, the array will be extended and any intermediate elements set to null.

The following example shows how to create a simple array with three values. Notice that they do not need to be added in sequence:

```
Ok = UDOCreate(UDO_OBJECT, hJSON)
Ok += UDOCreate(UDO_ARRAY, hArray)
Ok += UDOArraySetItem(hArray, 1, "first")
Ok += UDOArraySetItem(hArray, 3, "third")
Ok += UDOArraySetItem(hArray, 2, "second")
Ok += UDOSetProperty(hJSON,"array",hArray)

If Ok <> UDO_SUCCESS Then
   GoSub ShowError
End Else
   Ok = UDOWrite(hJSON, UDOFORMAT_JSON, OutputString)
   Crt OutputString
End
Ok = UDOFree(hJSON)
```

The result is:

```
{
    "array":      ["first", "second", "third"]
}
```

The flexibility in the model comes from the fact that any of the array elements can itself be another array or an object, and similarly any object property value can likewise be an array or object. This means you can create descriptive structures with any format, but as you may have noticed it is time consuming to write.

Lab 6.2

Create a second program named udo2.

This will request an author id and read the list of their titles from the BOOK_AUTHORS file. Build these into a document with the format below:

```
{
     "authorId":    "2",
     "name": "Rudyard Kipling",
     "titles":      [{
                "titleId":     "2",
                "title":       "Just So Stories",
                "price":       "14.99",
                "ISBN": "0955064007"
           }, {
                "titleId":     "207",
                "title":       "The Jungle Book: Selected Stories (Puffin Classics)",
                "price":       "7.99",
                "ISBN": "0140865543"
           }, {
                "titleId":     "210",
                "title":       "Just So Stories: Unabridged (Puffin Classics)",
                "price":       "7.99",
                "ISBN": "0140865551"
           }]
}
```

File Operations

These use a special series of buffers called File Buffers. These are limited in number with only 10 buffers available, each of which is associated at run time with a particular file. The pattern for file operations using a PROC is:

- A file is opened to a file buffer
- A record is read into the buffer
- The buffer contents are amended
- The buffer contents are written back to file

Real World

I would like to make it really clear at this point that doing file update operations from PROC is a generally bad idea. PROCs are not easily maintained, do not have centralized logic and because of their limited nature are prone to bypass any serious validation.

Only use updates from PROC if a) the nature of the update is truly trivial or b) as part of a bootstrap process where you may not have access to UniVerse Basic.

File buffers are referenced using an ampersand followed by the buffer number. Individual elements in the buffer, which effectively equate to the fields in a record, are referenced by number following a period:

```
&buffer.element
```

For example, the second field of the record held in the first file buffer:

```
T &1.2
```

You can also use indirect buffer references to the element, though this starts to become messy:

```
T &1.%2

T &1.%%2
```