
Notes for Sirius Mods Release 7.6

November 3, 2009



Sirius Software, Inc.
875 Massachusetts Avenue, Suite 21
Cambridge, MA 02139

Telephone: (617) 876-6677
FAX: (617) 234-1200
E-mail: support@sirius-software.com
World Wide Web: <http://sirius-software.com>

August 6, 2010

© 2010 Sirius Software, Inc.

Proprietary Notices

The following products:

- *Janus SOAP*
- *Janus Sockets*
- *Janus TCP/IP Base*
- *Janus Web Server*
- *Sirius Functions*
- *Sir2000 Field Migration Facility*

are proprietary products of Sirius Software, Inc.:

Sirius Software, Inc.
875 Massachusetts Avenue, Suite 21
Cambridge, Massachusetts 02139
USA

Model 204® and **Connect *™** are proprietary products of Computer Corporation of America, a wholly-owned subsidiary of Rocket Software, Inc., which owns the trademarks:

Rocket Software Corporate Office
M204 Division
275 Grove Street
Suite 3-410
Newton, Massachusetts 02466-2272
USA

SoftSpy™ is a proprietary product of Information Technology Systems:

Information Technology Systems
95 Wells Avenue
Newton, Massachusetts 02459-3216
USA

Contents

Proprietary Notices	ii
Contents	iii
Chapter 1: Introduction	1
Chapter 2: Maintenance and Support	3
Model 204 support	3
Chapter 3: All or Multiple Products	5
New SirMon statistics	5
Chapter 4: Janus SOAP ULI	7
XHTML entities for converting to Unicode	7
Print/Audit/Trace statements character-encode Unicode values	7
Codepages 1047EXT, 0037EXT, and 00285EXT on UNICODE command	8
ASCII translations with xxxEXT codepages	9
Migrating to codepage 1047EXT, 0037EXT, or 00285EXT	11
Unicode enhancement methods	12
Changes to Auto New and Allow Auto	13
Method-variable assignment restriction is dropped	14
Intrinsic classes	14
New Float method: FloatToString	14
New Float method: Round	15
New String method: Insert	16
New Unicode method: UnicodeWith	16
Collection classes	17
New system class: UnicodeNamedArraylist	17
Audit subroutine	19
Copy function	19
Count property	19
DeepCopy function	20
Default property	20
Item property	20
ItemByNumber property	20
Maximum function	21
Minimum function	21
NameByNumber property	21
New constructor	21

Number property	22
Print subroutine	22
RemoveItem function	22
Trace subroutine	23
UseDefault property	23
Print method for the collection classes	24
Searching methods for the collection classes	28
New defaults for collection maxima, minima, and sorting	31
File classes	32
Enhancements to Recordset and SortedRecordset instantiation	32
Inheritance support for Recordset and SortedRecordset objects	32
Targets of Find To, Sort To, Text To may execute a constructor	32
Targets of Find To, Sort To, Text To may be a class variable or property	33
New constructor for SortedRecordset and RecordsetCursor objects	33
New cursor state: Empty	35
Exception classes	35
New exception class: ItemNotFound	35
New constructor	36
New exception class: RecordLockingConflict	36
New exception class: XmlParseError	39
New utility class: SelectionCriterion	42
Declaring a SelectionCriterion object variable	44
Specifying a SelectionCriterion's parameters	44
The EQ, NE, GE, GT, LE, and LT constructors	47
The OR and AND constructors	49
The NOT constructor	50
The TRUE and FALSE constructors	50
Chapter 5: Janus SOAP Xml API	53
XmlDocs now maintained in Unicode	53
New methods	53
AllowNull XmlDoc property	53
AppendValue method in XmlNode class	54
NoEmptyElement property of element XmlNodes	55
Changes to existing methods	56
AllowUntranslatable deserialization option	56
CharacterEncodeAll option of Print, Audit, and Trace	57
InvalidChar property removed	58
InvalidCharacterPosition shared XmlDoc function	58
IsValidString shared XmlDoc function	59
Chapter 6: Janus TCP/IP Base	61
New forms of JANUS LOADXT	61

Chapter 7: Janus Sockets	63
SMTP Helper	63
New parameters for AddPart method	63
Chapter 8: Sir2000 Field Migration Facility	65
SIRFIELD command not allowed with FILEORG=X'100'	65
Chapter 9: Compatibility/Bug fixes	67
Backwards incompatibilities	67
Janus SOAP ULI	67
ToXmlDoc default changed to AttributeNames=True	67
Recordset LockStrength inconsistencies	67
Print/Audit/Trace statements encode Unicode ampersands	68
Janus SOAP Xml API	68
Serialization of attributes always bracketed by quotes	68
InvalidChar XmlDoc property now obsolete	69
Change to structure of LoadParameterInfo document	69
Fixes in Sirius Mods 7.6 but not in 7.5	70
Version corequisites	70

CHAPTER 1 ***Introduction***

This document lists the enhancements and other changes contained in *Sirius Mods* version 7.6, which was released in November 3, 2009. The previous version of the *Sirius Mods*, 7.5, was available in April, 2009.

CHAPTER 2 *Maintenance and Support*

2.1 Model 204 support

Sirius Mods version 7.6 supports only *Model 204* versions V7R1 and V6R1.

CHAPTER 3 *All or Multiple Products***3.1 New SirMon statistics**

These statistics are added for version 7.6. They are retrievable or viewable with \$SYSTAT or *SirMon*:

- JSCREENS** Counts the number of times *Janus Web Server* incremented the SCREENS statistic. Since these increments may occur outside of request context, they will not necessarily have corresponding Since-Last statistics. *Janus Web Server* will only increment SCREENS for connections that were closed with Status Code 200 (OK).
- JWEBERRS** Counts the number of times that *Janus Web Server* returned a 4xx status code. Note that these instances will not be counted in the SCREENS statistic, but it is useful to know that, for example, many 404 or 401 errors are occurring.
- SFTRSTRT** Counts the number of times a soft user restart has occurred.
- HRDRSTRT** Counts the number of times a hard user restart has occurred.

The following sections describe changes in the *Janus SOAP ULI* in this release.

4.1 XHTML entities for converting to Unicode

In addition to using hexadecimal character references, you can now use XHTML entities to represent Unicode characters when converting from EBCDIC to Unicode. Just like character references, XHTML entities are converted by the `U` constant method, or by the `CharacterDecode=True` argument on the `EbcDicToUnicode` and `EbcDicToAscii` methods.

For example, the following declaration initializes a Unicode %variable to the “non-breaking-space” character:

```
%nbsp Is Unicode Initial('&nbsp;':U)
```

You can find the list of XHTML entities on the Internet at the following URL:

```
http://www.w3.org/TR/xhtml1/dtds.html#h-A2
```

Except for the five predefined entities (ampersand, apostrophe, greater than, less than, and double quotation mark), the XHTML entities are still *not* acceptable to the *Janus SOAP XML* document deserializing methods.

4.2 Print/Audit/Trace statements character-encode Unicode values

The Print, Audit, and Trace statements have been changed so that when displaying Unicode data, any Unicode character not translatable to EBCDIC is displayed using the character encoding (“&#xhhhh;”) of the character, and any ampersand character is displayed using the string “&”. For example, the following fragment using the new `UnicodeWith` method (“[New Unicode method: UnicodeWith](#)” on page 16):

```
%u Unicode
%u = 'Trademark (&trade;):U:UnicodeWith(' & ')
%u = %u:UnicodeWith('Euro (&euro;):U)
Print %u
```

Results in:

```
Trademark (&#x2122;) &amp; Euro (&#x20AC;)
```

As described in “Print/Audit/Trace statements encode Unicode ampersands” on page 68, in the case of ampersand, this represents an upwards incompatibility.

4.3 Codepages 1047EXT, 0037EXT, and 00285EXT on UNICODE command

There are three new codepages that can be used in the UNICODE command:

- 1047EXT** Same as 1047, except that there are mappings between EBCDIC and Unicode for the 27 "extended" characters in the Microsoft 1252 enhanced version of ISO-8859-1.
- 0037EXT** Same as 0037, except that there are mappings between EBCDIC and Unicode for the 27 "extended" characters in the Microsoft 1252 enhanced version of ISO-8859-1.
- 0285EXT** Same as 0285, except that there are mappings between EBCDIC and Unicode for the 27 "extended" characters in the Microsoft 1252 enhanced version of ISO-8859-1.

To see the extended characters mapped by these codepages, issue, for example, the following command:

```
UNICODE Difference Codepages 0037 And 0037EXT
```

This will show the 27 extended mappings, for example:

```
*          Table 1 has Trans E=20 Invalid  
UNICODE Table Standard Map  E=20 Is U=20AC
```

This indicates that in codepage 0037, EBCDIC codepoint X'20' is not translatable to Unicode (nor is Unicode codepoint 20AC translatable to EBCDIC), while in codepage 0037EXT, these two codepoints are mapped to each other. U+20AC is the Unicode “Euro” character.

The codepoint mappings shown will be the same if you substitute “1047” or “0285” for “0037” in the above command.

In addition to providing the extended mappings between Unicode and EBCDIC, using any of 1047EXT, 0037EXT, or 00285EXT as the base codepage affects translations involving “ASCII”, as described in the following section.

4.3.1 ASCII translations with xxxEXT codepages

With “non-xxxEXT” codepages, Unicode characters correspond to “ASCII” characters with the same numeric value of the codepoint, for example, Unicode U+86 (the “Start Of Selected Area” control character) corresponds to the same ASCII control character at codepoint X'86'.

The Microsoft 1252 encodings redefine the mappings between “ASCII” and Unicode for the extended characters, as follows:

ASCII	Unicode
-----	-----
X'80'	U+20AC: Euro
X'82'	U+201A: Single comma quotation mark
X'83'	U+0192: Small letter script f
X'84'	U+201E: Double comma quotation mark
X'85'	U+2026: Horizontal ellipsis
X'86'	U+2020: Dagger
X'87'	U+2021: Double dagger
X'88'	U+02C6: Modifier letter circumflex
X'89'	U+2030: Per mille sign
X'8A'	U+0160: Capital letter S with caron
X'8B'	U+2039: Single left-pointing angle quote
X'8C'	U+0152: Capital ligature OE
X'8E'	U+017D: Capital letter Z with caron
X'91'	U+2018: Left single quotation mark
X'92'	U+2019: Right single quotation mark
X'93'	U+201C: Left double quotation mark
X'94'	U+201D: Right double quotation mark
X'95'	U+2022: Bullet
X'96'	U+2013: En dash
X'97'	U+2014: Em dash
X'98'	U+02DC: Small tilde
X'99'	U+2122: Trademark sign
X'9A'	U+0161: Small letter s with caron
X'9B'	U+203A: Single right-pointing angle quote
X'9C'	U+0153: Small ligature oe
X'9E'	U+017E Small letter z with caron
X'9F'	U+0178 Capital letter Y with diaeresis

To keep the implicit translations between Unicode and “ASCII” invertible when any of 1047EXT, 0037EXT, or 00285EXT is the base codepage, the Unicode character with the same numerical value as any of the above ASCII codepoints is **not** translatable to ASCII. For example, U+9F would not be translatable to ASCII.

Using any of 1047EXT, 0037EXT, or 00285EXT as the base codepage affects translations involving “ASCII”:

- Translations performed by the EbcdicToAscii method

If an EBCDIC codepoint (for example, X'20' in the base) maps to one of the extended characters (U+20AC), that codepoint will map to the "ASCII" codepoint to which the Unicode character maps with Microsoft 1252 (U+20AC maps to "ASCII" X'80'). Therefore, given the following input:

```
UNICODE Table Standard Base Codepage 0037EXT
Begin
PrintText {$X2C('20'):EbcDicToAscii:StringToHex}
End
```

The result is:

```
80
```

Note: As often is the case when explaining various features of Unicode support, an example shows a UNICODE command to make explicit the translations being used. In practice, the UNICODE command should only be issued during *Model 204* initialization.

- Translations performed by the AsciiToEbcDic method

An ASCII codepoint will map to EBCDIC by, in effect:

1. Translating the ASCII codepoint to Unicode using the Microsoft 1252 mapping
2. Translating that Unicode character to EBCDIC as would the UnicodeToEbcDic function

- Translation from "ASCII" to Unicode when deserializing an XML document with the `encoding="ISO-8859-1"` declaration

If any of 1047EXT, 0037EXT, or 00285EXT is the base codepage, the Microsoft 1252 mappings are used to convert ASCII to Unicode. For example, given the following input:

```
UNICODE Table Standard Base Codepage 0037EXT
Begin
%doc Object XmlDoc Auto New
%s Longstring
%s = '<?xml version="1.0" encoding="ISO-8859-1"?>' -
  With '<x>'
%s = %s:EbcDicToAscii
%s = %s With '80':HexToString
%s = %s With '</x>':EbcDicToAscii
%doc:LoadXml(%s)
Print %doc:Value:StringToHex
End
```

The result is:

20

The result occurs because the ASCII X'80' input is translated to U+20AC using the Microsoft 1252 mappings, and the Print statement translates U+20AC to EBCDIC X'20' using the Unicode to EBCDIC mappings in codepage 0037EXT. If codepage 0037 were used, the request would be cancelled with a parsing error, because the X'80' ASCII/Unicode character is a control character that is not allowed by the XML standard to be deserialized into an XML document.

4.3.2 Migrating to codepage 1047EXT, 0037EXT, or 00285EXT

If you find that some of your XML document processing is unsuccessful because it contains some of the Unicode characters listed in [“ASCII translations with xxxEXT codepages” on page 9](#), you may benefit by switching your base codepage, for example, from 0037 to 0037EXT.

The principal effect of switching will be to allow the set of 27 Unicode characters, 26 of which were previously untranslatable to EBCDIC. Because one of these mappings (U+85) was translatable to EBCDIC (X'15'), you may see the following subtle differences using these codepages, compared to using their “non-EXT” counterparts (without any further modifications using the UNICODE command):

- The EbcDicToAscii function, when an input character is X'15', results in an untranslatable character exception, rather than producing the X'85' ASCII Next Line control character. (Note that the mapping between EBCDIC X'15' and U+0085 is unchanged.)
- The AsciiToEbcDic function, when an input character is X'85', results in the X'21' EBCDIC character, rather than the X'15' character.
- If you are deserializing an ASCII XML document with the `encoding="ISO-8859-1"` declaration, and that document contains the ASCII X'85' character, then the X'85' is treated as the horizontal ellipsis character, rather than the “next line” control character.

4.4 Unicode enhancement methods

Enhancement methods for Unicode type string objects are now allowed. Although the User Language Unicode type was introduced in *Sirius Mods* version 7.3, enhancement methods for Unicode objects have not been allowed until this release.

For example, prior to *Sirius Mods* version 7.6., a local function like the following was invalid:

```
local function (unicode):foobar is float
    ...
end function
```

Similarly, a function like the following inside a class was invalid:

```
class util
    public
        function (unicode):foobar is float
            ...
        end class
```

This restriction is now removed, and you can define an enhancement method like the following:

```
begin

local function (unicode):unicodeReverse is unicode
    %result is unicode
    %i is float
    for %i from %this:unicodeLength to 1 by -1
        %result = -
            %result:unicodeWith(%this:unicodeChar(%i))
    end for
    return %result
end function

%u is unicode

%u = 'Bye-bye, Miss American &pi;':u
printText {~} = "{%u}", {~} = "{%u:unicodeReverse}"

end
```

This request produces the following result:

```
%u = "Bye-bye, Miss American &#x03C0;"
%u:unicodeReverse = "&#x03C0; naciremA ssiM ,eyb-eyB"
```

4.5 Changes to Auto New and Allow Auto

If you append `Auto New` to an object variable declaration for a user-defined class object, you are *not* required to later provide a statement that explicitly instantiates the object. Specifying `Auto New` is valid only if the class definition includes `Allow Auto`.

As of version 7.6 of the *Sirius Mods*, the `Auto New` functionality is allowed for extension classes — as long as all the base classes are defined as `Allow Auto`.

In addition, a related enhancement is added to `Allow Auto`. Prior to this release, a class defined with `Allow Auto` was not allowed to contain a constructor. As of version 7.6 of the *Sirius Mods*, `Allow Auto` is allowed in a class that has a constructor. However, for an object variable declared with `Auto New` in such a class, an automatic instantiation of the object will **not** call any of the constructors in the class, even if the class has defined an explicit constructor with the name `New`.

The `Auto New` and `Allow Auto` changes are shown in this example. The following request completes successfully, and class `Y`'s `New` constructor does **not** get called by `%y:b`:

```

b
class X
  public
    allow auto
    variable a is float
  end public
end class

class Y extends x inherit
  public
    allow auto
    variable b is float
    constructor new
  end public

  constructor new
    print 'con y'
    construct %(x):new
  end constructor
end class

%y is object y auto new
print and %y:b
end

```

The result is:

```

∅

```

4.6 Method-variable assignment restriction is dropped

As of *Sirius Mods* version 7.6, you may assign an overridable method to a method variable. Prior to this version, such an assignment was not allowed.

4.7 Intrinsic classes

4.7.1 New Float method: FloatToString

This function converts a floating point number to a string with a specific length and number of decimal places.

```
%str = number:FloatToString([Length=len] [, Dp=num])
```

FloatToString syntax

Notes:

- If the target length and number of decimal places would result in leading non-zero digits being lost, the request is canceled. For example, specifying the following results in request cancellation:

```
13.1415926:floatToString(length=6, dp=4)
```
- FloatToString neatly displays non-integer numeric values. The Round method ([“New Float method: Round” on page 15](#)) is not a good way of doing this, since Round returns a Float value and strips trailing 0s when printing the value.
- FloatToString currently does not support E format (9.9E72) output.
- A Dp value of 0 has a special result: no decimal point. There is no way to get a trailing decimal point without any digits after it (unless you explicitly append a point character (.) to a result for which Dp=0 is specified. As is shown in the last of the PrintText statements in the example below, you **can** get a value with a leading decimal point and no digit before it.
- To produce the specified length or number of decimal places in the result, FloatToString uses truncation, not rounding, of the input, as you can see in the last two of the PrintText statements in the example below.

The following program demonstrates the FloatToString method.

```
begin
  printText {~} = {0.1415926:floatToString}
  printText {~} = {0.1415926:floatToString(dp=0)}
  printText {~} = {0.1415926:floatToString(dp=1)}
  printText {~} = {0.1415926:floatToString(dp=5)}
  printText {~} = {0.1415926:floatToString(length=10)}
  printText {~} = {0.1415926:floatToString(length=10, dp=1)}
  printText {~} = {0.1415926:floatToString(length=10, dp=4)}
  printText {~} = {0.1415926:floatToString(length=5, dp=4)}
end
```

The result is:

```
0.1415926:floatToString = 0
0.1415926:floatToString(dp=0) = 0
0.1415926:floatToString(dp=1) = 0.1
0.1415926:floatToString(dp=5) = 0.14159
0.1415926:floatToString(length=10) = 0
0.1415926:floatToString(length=10, dp=1) = 0.1
0.1415926:floatToString(length=10, dp=4) = 0.1415
0.1415926:floatToString(length=5, dp=4) = .1415
```

4.7.2 New Float method: Round

This function returns a floating point number that is the method object number rounded to a specified number of decimal places.

```
%num = number:Round(dp)
```

Round syntax

Negative numbers are rounded down to the nearest integer; positive numbers are rounded up.

The following program demonstrates the Round method.

```
begin
  printText {~} = {3.1415926:round}
  printText {~} = {3.1415926:round(0)}
  printText {~} = {3.1415926:round(1)}
  printText {~} = {3.1415926:round(4)}
end
```

The result is:

```
3.1415926:round = 3
3.1415926:round(0) = 3
3.1415926:round(1) = 3.1
3.1415926:round(4) = 3.1416
```

4.7.3 New String method: Insert

This function inserts an argument string inside the method object string, starting before the specified position in the method object string.

```
%outStr = string:Insert(%insrtString, %before)
```

Insert syntax

The value of *%before* must be between 1 and the number of characters in the method object string plus one. An invalid position produces a request cancellation.

The following request contains four Insert method calls:

```
Begin
printText {~} = {'':insert('xyz', 1)}
printText {~} = {'ABC':insert('xyz', 3)}
printText {~} = {'ABC':insert('xyz', 4)}
printText {~} = {'ABC':insert('xyz', 5)}
End
```

The request result is:

```
':insert('xyz', 1) = xyz
'abc':insert('xyz', 3) = ABxyzC
'abc':insert('xyz', 4) = ABCxyz
'abc':insert('xyz', 5) =
*** 1 CANCELLING REQUEST: MSIR.0750: Class STRING,
function INSERT: insertion position greater than length
of input string plus one in line 5
```

4.7.4 New Unicode method: UnicodeWith

This function returns the Unicode string that is the concatenation of the method object Unicode string and its Unicode string argument.

```
%outUni = unicode:UnicodeWith(unistring)
```

UnicodeWith syntax

For example, the following is a simple UnicodeWith call:

```
Begin
  %u is unicode Initial('This is an at sign: ')
  Print %u:unicodeWith('&#x40;':U)
End
```

The result is:

```
This is an at sign: @
```

4.8 Collection classes

4.8.1 New system class: UnicodeNamedArraylist

The UnicodeNamedArraylist class is nearly identical to the NamedArraylist class. The main difference is that instead of EBCDIC subscript names for items as in the NamedArraylist class, the name subscripts in a UnicodeNamedArraylist object are Unicode values. UnicodeNamedArraylist items are stored by item name in Unicode order, whereas NamedArraylist items are stored by item name in EBCDIC order.

Note: The names of UnicodeNamedArraylists are limited to 127 characters (versus 255 bytes for NamedArraylists).

The following annotated request demonstrates the methods where the distinction between a UnicodeNamedArraylist and a NamedArraylist is significant. The U constant method handily creates Unicode strings in the item name subscripts. Those names also begin with an XHTML entity reference.

```
b

%i is float
%k is unicodeNamedArraylist of longstring
%m is Arraylist of longstring
%k = new

* implicit Item method sets list item values:
%k('&sect;Jan':u) = 'Orion'
%k('&sect;Apr':u) = 'Leo'
%k('&sect;Mar':u) = 'Cancer'
%k('&sect;Jun':u) = 'Ursa Minor'
%k('&sect;Nov':u) = 'Andromeda'
%k('&sect;Dec':u) = 'Aries'
%k('&sect;Feb':u) = 'Canis Major'

* print the value of the item whose name is the
* Item method argument:
print %k:Item('&sect;Jan':u)
```

```
* print item number of named item:
print %k:number('&sect;Nov':u)

* print name of each list item, in order of item position
* from first to last:
for %i from 1 to %k:count
    print %i ': ' %k:nameByNumber(%i)
end for

* how many items remain after specified item is removed?
printText {~} = {%k:removeItem('&sect;Dec':u)}

* names and values of list items in order by item position:
for %i from 1 to %k:count
    print %k:nameByNumber(%i) ': ' and %k:itembyNumber(%i)
end for

* sort list values alphabetically in descending order
%m = %k:sortNew(descending(this))
%m:print

end
```

The request prints:

```
Orion
7
1: $Apr
2: $Dec
3: $Feb
4: $Jan
5: $Jun
6: $Mar
7: $Nov
%k:removeItem('&sect;Dec':u) = 6
$Apr: Leo
$Feb: Canis Major
$Jan: Orion
$Jun: Ursa Minor
$Mar: Cancer
$Nov: Andromeda
1: Ursa Minor
2: Orion
3: Leo
4: Canis Major
5: Cancer
6: Andromeda
```

The individual UnicodeNamedArraylist methods are described briefly in the following subsections.

4.8.1.1 Audit subroutine

This subroutine displays the contents of a UnicodeNamedArraylist in a readable form, useful for debugging, for example. Audit is identical to the Print method (“[Print subroutine](#)” on page 22), except the result is sent to the *Model 204* audit trail (like the User Language Audit statement).

```
[%rc =] %unamrayl:Audit( [method]           -
                        [, NumWidth=itemnumlen] -
                        [, Separator=separator] -
                        [, NameWidth=itemnamelen] -
                        [, Start=firstitem]     -
                        [, MaxItems=maxitems]   -
                        [, Label=label] )
```

Audit syntax

4.8.1.2 Copy function

This function makes a “shallow” copy of the UnicodeNamedArraylist method object *%unamrayl*. If *%unamrayl* contains objects, they are **not** copied. If *%unamrayl* is Null, a Null is returned.

```
%cop = %unamrayl:Copy
```

Copy syntax

4.8.1.3 Count property

This ReadOnly property returns the number of items in the UnicodeNamedArraylist method object.

```
%num = %unamrayl:Count
```

Count syntax

4.8.1.4 **DeepCopy function**

This function makes a “deep copy” of the UnicodeNamedArraylist method object, `%unamrayl`. If `%unamrayl` contains objects, they **are** copied. If `%unamrayl` is Null, a Null is returned.

```
%dcop = %unamrayl:DeepCopy
```

DeepCopy syntax

4.8.1.5 **Default property**

This ReadWrite property indicates the value to be returned if a requested item name is not in the UnicodeNamedArraylist and the UseDefault property (“UseDefault” on page 23) is set to True.

```
%val = %unamrayl:Default  
  
%unamrayl:Default = %val
```

Default syntax

4.8.1.6 **Item property**

This ReadWrite property returns or sets the value of the item that has the specified subscript name in the UnicodeNamedArraylist.

```
%item = %unamrayl:Item(name)  
  
%unamrayl:Item(name) = %item
```

Item syntax

4.8.1.7 **ItemByNumber property**

This ReadWrite property returns or sets the UnicodeNamedArraylist item that has the specified item number. Item number is ordinal, so ItemByNumber(3), for example, identifies the third item in the ordered-by-name UnicodeNamedArraylist.

```
%item = %unamrayl:ItemByNumber(number)  
  
%unamrayl:ItemByNumber(number) = %item
```

ItemByNumber syntax

4.8.1.8 Maximum function

This function returns the subscript name of the UnicodeNamedArraylist item that has the maximum value after the application of a specified function to each item. The function that gets applied to each UnicodeNamedArraylist item, which you identify in the argument to Maximum, must be a method that operates on the item type and returns a User Language intrinsic datatype (Float, String, Longstring, or Unicode) value.

```
name = %unamrayl:Maximum(function)
```

Maximum syntax

4.8.1.9 Minimum function

This function returns the name (subscript) of the UnicodeNamedArraylist item that has the minimum value after the application of a specified function to each item. The function that gets applied to each UnicodeNamedArraylist item, which you identify in the argument to Minimum, must be a method that operates on the item type and returns a User Language intrinsic datatype (Float, String, Longstring, or Unicode) value.

```
name = %unamrayl:Minimum(function)
```

Minimum syntax

4.8.1.10 NameByNumber property

This ReadOnly property returns the subscript name of the item that has the specified item number (position) in the UnicodeNamedArraylist.

```
name = %unamrayl:NameByNumber(number)
```

NameByNumber syntax

4.8.1.11 New constructor

This method returns a new instance of a UnicodeNamedArraylist.

```
%unamrayl = New
```

New syntax

4.8.1.12 **Number property**

This ReadOnly property returns the item number (ordinal) of the item that has the specified subscript name in the UnicodeNamedArraylist.

```
%num = %unamrayl:Number(name)
```

Number syntax

4.8.1.13 **Print subroutine**

This method displays the contents of a UnicodeNamedArraylist on the user's standard output device, typically a terminal. The list item values, displayed in order by their subscript names, are preceded by their item number and item name, both of which by default are followed by a colon (:) and a blank.

```
[%rc =] %unamrayl:Print( [method]           -  
                        [, NumWidth=itemnumlen] -  
                        [, Separator=separator] -  
                        [, NameWidth=itemnamelen] -  
                        [, Start=firstitem]     -  
                        [, MaxItems=maxitems]   -  
                        [, Label=label] )
```

Print syntax

4.8.1.14 **RemoveItem function**

This callable function removes from the UnicodeNamedArraylist the item that has the specified subscript name.

```
[%num =] %unamrayl:RemoveItem(name)
```

RemoveItem syntax

4.8.1.15 Trace subroutine

This subroutine displays the contents of a UnicodeNamedArraylist in a readable form, useful for debugging, for example. Trace is identical to the Print method ([“Print subroutine”](#)), except the result is sent to the selected Trace stream (like the User Language Trace statement).

```
[%rc =] %unamrayl:Trace( [method]           -
                        [, NumWidth=itemnumlen] -
                        [, Separator=separator] -
                        [, NameWidth=itemnamelen] -
                        [, Start=firstitem]     -
                        [, MaxItems=maxitems]   -
                        [, Label=label] )
```

Trace syntax

4.8.1.16 UseDefault property

This ReadWrite property indicates whether an attempted retrieval of an item that is not on the UnicodeNamedArraylist should return the Default method value ([“Default” on page 20](#)).

UseDefault will return, or may be assigned, only the values `True` or `False`. Its initial value is `False`.

```
%bool = %unamrayl:UseDefault
%unamrayl:UseDefault = %bool
```

UseDefault syntax

4.8.2 Print method for the collection classes

In versions of the *Sirius Mods* prior to 7.6, the standard way to view the entire contents of a collection is to loop through the list items and display each one using a User Language Print statement (or Audit or Trace). For a NamedArraylist, for example, you use a method for the item subscript name and a method for the item content:

```
%nal is namedArraylist of float
...

%nal = new
%nal('Chicago') = 22
%nal('New York') = -999
%nal('Los Angeles') = 3.1415926
%nal('Philadelphia') = 1099

for %i from 1 to %nal:count
  print %nal:nameByNumber(%i) and %nal:itemByNumber(%i)
end for
```

This is the result:

```
Chicago 22
Los Angeles 3.1415926
New York -999
Philadelphia 1099
```

As of version 7.6 of the *Sirius Mods*, the Print method for any collection does the work of the loop in the preceding example, and more. Supplied for debugging purposes, Print (or the essentially identical Audit or Trace method) would produce the following output using the example collection above (that is: %nal:print):

```
1: Chicago: 22
2: Los Angeles: 3.1415926
3: New York: -999
4: Philadelphia: 1099
```

Notice that Print outputs all the collection items (or, optionally, a range of items), and it also includes:

- The ordinal, or position, number for each item
- A separator string after the item position number and also after the item name (if a named collection)

Print also has optional parameters that let you specify:

- The lengths for the item name and number
- A label string to precede each output line
- The number of items to display

The Print method applies a ToString method (by default) to each item value (and always to each item name), to produce its result. Applying Print to a collection whose item types are *not* system classes will work only if the user class contains a ToString method.

The general syntax of Print (Audit or Trace) for a collection is:

```
%coll:Print (<method>, <numWidth>, <nameWidth>, -  
            <separator>, <start>, <maxItems>, <label>)
```

All parameters are optional and all except <method> have required names (which match the names used in the syntax above). The parameters are described briefly below and in greater detail in the method descriptions for the appropriate collection type in the ***Janus SOAP Reference Manual***.

- <method>** The method applied to collection items to produce the printed output. The method must take no parameters and produce an intrinsic (Float, String, Fixed, Unicode) value. It may be a system or user-written method, a class Variable or Property, a local method, or a method variable. The default is the ToString method.
- <numWidth>** The number of bytes for the item number in the output. If 0, the default, the item number is not printed.
- <nameWidth>** The number of bytes for the item name (ignored if an Arraylist). If -1, the default, the entire name is fit exactly. If 0, the item name is not printed.
- <separator>** A string that follows the item number and that repeats after the item name. The default is a colon. A blank follows each instance of *separator*.
- <start>** The number of the collection item from which to start the output display. By default, the display begins from item one.
- <maxItems>** The maximum number of collection items to print. By default, all items are displayed.
- <label>** A string, null by default, marking the beginning of each item's line of output.

Additional Print examples follow:

1. For the NamedArraylist in the example [on page 24](#), but issuing `%nal:print(numWidth=3, nameWidth=14)`, this is the result:

```
1: Chicago      : 22
2: Los Angeles  : 3.1415926
3: New York     : -999
4: Philadelphia : 1099
```

If you issue `%nal:print(numWidth=3, nameWidth=7)`, the result is:

```
1: Chicago: 22
2: Los Ang: 3.1415926
3: New Yor: -999
4: Philade: 1099
```

You can define a local function to display your output:

```
local function (string):quote is longstring
  return ''' with %this with '''
end function
```

Now you issue:

```
%nal:print(quote)
```

And you get this result:

```
1: Chicago: '22'
2: Los Angeles: '3.1415926'
3: New York: '-999'
4: Philadelphia: '1099'
```

If you named your local method `Tostring` instead of `Quote`, it would not need to be specified on the Print method. This is shown in the following example.

2. In the following request, the method parameter used with the arraylist Print method is a class Variable:

```

b
class python
  public
    variable surname      is string len 30
    variable givenName    is string len 30
    variable routine      is string len 30
    constructor new(%surname is string len 30, -
                    %givenName is string len 30, -
                    %routine is string len 30)

  end public

  constructor new(%surname is string len 30, -
                  %givenName is string len 30, -
                  %routine is string len 30)
    %this:surname = %surname
    %this:givenName = %givenName
    %this:routine = %routine
  end constructor
end class

%pythons is arraylist of object python

%pythons = list(new('Cleese', 'John', 'Dead Parrot'), -
               new('Palin', 'Michael', 'Lumberjack'), -
               new('Idle', 'Eric', 'Nudge nudge'), -
               new('Chapman', 'Graham', 'Throat Wobbler Mangrove'), -
               new('Jones', 'Terry', 'Mouse Organ'))

%pythons:print(surname)
end

```

The List method used above is described in the *Janus SOAP Reference Manual*.
The request result is:

```

1: Cleese
2: Palin
3: Idle
4: Chapman
5: Jones

```

If you create the following ToString method in the class:

```

function toString is longstring
  return 'surname=' with %this:surname with ', ' with -
         'givenName=' with %this:givenName with ', ' with -
         'routine=' with %this:routine
end function

```

And you issue this Print method call, which implicitly invokes your ToString method:

```
%pythons:print(start=2, maxItems=3)
```

The result is:

```
2: surname=Palin, givenName=Michael, routine=Lumberjack
3: surname=Idle, givenName=Eric, routine=Nudge nudge
4: surname=Chapman, givenName=Graham, routine=Throat
   Wobbler Mangrove
```

3. The ToString method, which is always applied to item name subscripts, even if you define a local or class ToString method, leaves names as is for NamedArraylists. For FloatNamedArraylists, it displays the float subscripts as strings. For UnicodeNamedArraylists, it translates names from Unicode to EBCDIC, (character-entity-encoding any non-translatable characters), as in the following:

```
%transcendental    is unicodeNamedArraylist of float
...
%transcendental = new
%transcendental('&pi;':U) = 3.1415926
%transcendental:print
```

The U constant method used above is described in the **Janus SOAP Reference Manual**. The Print result shows the encoded form of the Unicode item name:

```
1: &#x03C0;: 3.1415926
```

Note: If you specified %transcendental:print(namewidth=4), for example, the item name **is** truncated:

```
1: &#x: 3.1415926
```

4.8.3 Searching methods for the collection classes

As of *Sirius Mods* version 7.6, a variety of methods are common to all the collection classes for the purpose of searching a collection for the item(s) that satisfy one or more specified conditions.

The searching methods (all functions, listed below) have the same, or nearly the same syntax. They take two parameters:

- An object that specifies the search conditions (a `SelectionCriterion` object, added in *Sirius Mods* 7.6).
- A parameter (`Start`) that specifies where in the collection to begin the search. One method, `SubsetNew`, does not accept this parameter.

The searching methods are:

FindNextItem	Searching “forward” in the collection, finds the next item that matches a criterion, and returns that item.
FindPreviousItem	Searching “backward” in the collection, finds the next item that matches a criterion, and returns that item.
FindNextItemNumber	Searching “forward,” finds the next item that matches a criterion, and returns that item number.
FindPreviousItemNumber	Searching “backward,” finds the next item that matches a criterion, and returns that item number.
SubsetNew	Returns a new collection that contains all the items in the input collection that match the criterion.

The FindNextItem and FindPreviousItem methods also throw an `ItemNotFound` exception (“[New exception class: ItemNotFound](#)” on page 35) if no item matches the SelectionCriterion.

A SelectionCriterion object (“[New utility class: SelectionCriterion](#)” on page 42), which might consist of multiple components, describes a single selection criterion. For example, the `GE` method in that class uses two parameters to form a (“greater than or equal to”) comparison criterion to apply to the collection items. So, for SelectionCriterion object `%sel`, which selects items whose absolute value is less than or equal to 1000, you might have:

```
%sel = ge(absolute, 1000)
```

A simple search, starting from the eighth item in the `%payoff` Arraylist, might be:

```
%item = %payoff:findNextItem(%sel, start=7)
```

The parameters of the SelectionCriterion `GE` method above provide the operands for the comparison operator `GE`. In this case, `absolute`, is an intrinsic Float method which is applied to an item value. In general, this must be a function that operates on the type of the items in the collection, and it may be a local method or method variable or a class member (variable, property).

The value that results from applying the Absolute method above is compared to the second `GE` parameter, 1000. This 1000 may be any User Language intrinsic expression, such as a string or numeric literal.

In the fragment that follows, the function in the SelectionCriterion is a local method, and the searching method, `FindPreviousItemNumber`, searches backward starting with

the tenth item in the collection to find the item number of the first item that satisfies the criterion:

```
%flt is arraylist of float
%sel is object selectionCriterion for float

local function (float):myMod is float
    return %this:mod(7)
end function

%sel = LT(myMod, 1)
%num = %flt:findPreviousItemNumber(%sel, start=11)
```

The local method `myMod` above, which calls the `Mod` intrinsic Float method, is necessary in this case because the `SelectionCriterion` function parameter may not itself specify a parameter. The function parameter is a method value, not a User Language expression.

The preceding example also shows a `SelectionCriterion` object declaration, which must suit the item type to which the criterion will be applied, as described in [“Declaring a SelectionCriterion object variable” on page 44](#).

In the following example, the function parameter is the very useful identity function, `This`, which returns the value of the item to which it is applied. The searching method `SubsetNew` returns a collection of all the items in the collection that satisfy either of the criteria (`< 0`, `> 999`) that comprise the `OR` criterion:

```
%sel = OR(LT(this, 0), GT(this, 999))
%arraylist = %flt:findPreviousItemNumber(%sel, start=11)
```

Usage notes:

- The main benefit of these searching methods is the ease of coding provided by their simplicity and flexibility. However, the `Find` and `Subset` operations on collections of objects will necessarily be considerably more expensive than the comparable operations on `$lists` or `Stringlists`. For example, a level of indirection between object references and objects makes the processing much more complicated than that for `Stringlists`. However, because the cost of `locates` or `subsets` is likely to be a small fraction of the cost of most applications, switching to objects for these applications offers the benefits of cleaner code without a major expense.
- The `FindNextItem` and `FindPreviousItem` methods throw an `ItemNotFound` exception if no item matches the `SelectionCriterion`, but the `FindNextItemNumber` and `FindPreviousItemNumber` methods *do not* throw an exception in that case. The following are the suggested guidelines for using these methods:
 - For simply checking if an item in a collection matches a `SelectionCriterion`, use `FindNextItemNumber` or `FindPreviousItemNumber`.

- For looping over a collection, use `FindNextItemNumber` or `FindPreviousItemNumber` with an `If` test.
- For extracting a single item that you are very sure must be in the collection, use `FindNextItem` or `FindPreviousItem`. If you are wrong about the presence of the item, the exception is thrown and the error is caught.
- For conditionally extracting a single item from a collection, use `FindNextItem` or `FindPreviousItem` with a `Try/Catch` clause.

4.8.4 New defaults for collection maxima, minima, and sorting

The special identity function, `This`, is now the default function parameter value for the collection `Minimum` and `Maximum` methods and for the `SortOrder Ascending` and `Descending` methods. In addition, for the `SortOrder` argument in the sorting methods, `Ascending` (which implies `Ascending(this)`) is now the default.

For example, instead of explicitly specifying `This` in the `printText` statement in this request:

```
b
%1 is arraylist of float
%1 = list(7, 4, -3, 11, 5)
printText {~} = {%1:maximum(this)}
end
```

You can now use:

```
printText {~} = {%1:maximum}
```

Similarly, for a sort, instead of `%1:sort(descending(this))`, you can now simply specify:

```
%1:sort(descending)
```

Furthermore, because `Ascending` is now the default argument, explicitly specifying `ascending(this)` in the following request is now optional:

```
b
%1 is arraylist of float
%1 = list(7, 4, -3, 11, 5)
%1:sort(ascending(this))
end
```

Instead, you can now simply specify:

```
%1:sort
```

4.9 File classes

4.9.1 Enhancements to Recordset and SortedRecordset instantiation

4.9.1.1 Inheritance support for Recordset and SortedRecordset objects

Sirius Mods version 7.6 supports Find To and Sort To statements whose target is an object of an extension class of either a Recordset or a SortedRecordset. That is, a sequence of statements like the following will now succeed:

```
class truckWrecks extends recordset in file trucks
...
end class
...
%trex is object truckWrecks
...
fd to %trex
...
```

4.9.1.2 Targets of Find To, Sort To, Text To may execute a constructor

In *Sirius Mods* version 7.6, a Find To or Sort To statement that constructs a Recordset object may execute a separate constructor method for the target variable before finding records for the target. For example, a statement like the following, which creates the already-declared Recordset %trex, is now valid:

```
fd to %trex = new(, LoopLockStrength=exclusive)
...
end find
```

In the statement above, Find To first creates a new Recordset instance (with the specified For loop lockstrength) and assigns it to the target object variable, %trex. Then Find To populates %trex with the found records.

Note: If the Find statement in the previous example is changed to this:

```
fdwol to %trex = new(, LoopLockStrength=exclusive)
```

The locking in the %trex Recordset is ultimately governed by the Find statement, so the resulting lockstrength here is None in agreement with Find Without Locks.

Similarly, the target of a `Text To` statement may now execute a constructor before populating the target list. For example, the following is now valid:

```
%mylist is object stringlist
text to %mylist = new
...
end text
```

Previously, you would have to issue one more statement:

```
%mylist is object stringlist
%mylist = new
text to %mylist
...
end text
```

4.9.1.3 Targets of Find To, Sort To, Text To may be a class variable or property

The Find To, Sort To, and Text To statements in *Sirius Mods 7.6* all allow their target to be a class variable or even a class property. Before this version, none of these were allowed, except Text To allowed the target to be a class variable.

Note: The property support allows the target of a Find To to be a collection member. For example, the following fragment outlines a statement sequence that would validly instantiate the Recordset object that is the item named for `%value` in the `%wrek` NamedArraylist:

```
%wrek is namedArraylist of object recordset in file foobar
...
fd to %wrek(%value)
  foo = %value
...
end find
```

4.9.2 New constructor for SortedRecordset and RecordsetCursor objects

Prior to this release, the SortedRecordset and RecordsetCursor classes had no constructor methods. Such objects could only be created with factory methods or their equivalents. *Sirius Mods* version 7.6 introduces simple New constructors for both classes:

- A SortedRecordset **New** constructor has no parameters and simply instantiates an empty instance of its class. For example:

```
%srs is object sortedRecordset in sordid
...
%srs = new
```

Although such a new instance is not of significant value, the New constructor is useful to enable the creation of an extension class of the SortedRecordset class, which the existing SortedRecordset constructor factory methods cannot do. See, for example, this use of New in a Construct statement in an extension class:

```
class sordidSet extends sortedRecordset in sordid inherit
...
  constructor new
    construct %(sortedRecordset in sordid):new
  ...
end constructor
...
end class
```

- Since a RecordsetCursor references a Recordset or SortedRecordset object and may have a LoopLockStrength, the RecordsetCursor **New** constructor is of this form:

```
%(RecordsetCursor):New(<recset> [, loopLockStrength=<lls>])
```

Where:

<recset> A required Recordset or SortedRecordset object.

<lls> A LockStrength enumeration setting the minimum lock strength for a record in a For Record At loop on a RecordsetCursor object.

Note: It is an error to specify this argument if **<recset>** is a SortedRecordset object.

If **<recset>** is empty, the RecordsetCursor New constructor returns a RecordsetCursor object with the state **Empty**, which is also new in *Sirius Mods* version 7.6.

Note: This is different from the Recordset and SortedRecordset **Cursor** methods, which construct a RecordsetCursor object. Those Cursor methods still return a null if **<recset>** is empty.

As is the case for the SortedRecordset New constructor described earlier, the RecordsetCursor New constructor enables the creation of an extension class of the RecordsetCursor class, which the existing RecordsetCursor-object factory constructor cannot do.

4.9.3 New cursor state: Empty

A RecordsetCursor cursor is in any of multiple positions, or *cursor states*, as it navigates a record set. *Sirius Mods* version 7.6 adds the `Empty` state to the set of possible cursor states. A cursor is in the Empty cursor state if it points to an empty record set.

The availability of the Empty state allows the `New` constructor (“[New constructor for SortedRecordset and RecordsetCursor objects](#)” on page 33) to avoid returning a null if it is applied to a record set that has no records. And this becomes important if you are constructing an extension of the RecordsetCursor or SortedRecordset class.

4.10 Exception classes

4.10.1 New exception class: ItemNotFound

An ItemNotFound exception indicates that a collection object search located no collection items that satisfy the selection criterion specified in the collection method that invoked the search. There are several searching methods (“[Searching methods for the collection classes](#)” on page 28), but only those that return a single found item produce an ItemNotFound exception.

This exception class has no properties. It is simply a notification that a valid search found no items that met the selection criterion.

The class's only method is the `New` constructor, which you would typically use with a User Language Throw statement to produce an ItemNotFound exception:

```
throw %(itemNotFound):new
```

When working with the collection searching methods, remember that an exception is thrown only if there is a catcher for the exception. For example, if you expect one or no matches from your search, you might specify a block like the following:

```
try %terminationInfo = -
    %terminationList:findNextItem (eq(custId, %custid)
    ... process the termination info
catch itemNotFound
end try
```

On the other hand, if you always expect a match, you might want a program crash to result if you are wrong about the match, so you could leave out a Try/Catch.

The Try/Catch approach is slightly more efficient than a test for a zero result. Try generates no quads other than the branch around the catch block if the search succeeds. And in fact, you could put a Try around a loop and use the ItemNotFound exception to exit the loop:

```
%i = 0
try
  repeat forever
    %i = %colla:findNextItemNumber(<criteria>), start=%i
    %obj = %colla(%i)
    ... process the object
  end repeat
catch itemNotFound
end try
```

4.10.1.1 **New constructor**

This callable constructor generates an instance of an `ItemNotFound` exception. The New method format follows:

<code>[%itemnf =] [%(<code>ItemNotFound</code>):]New</code>

ItemNotFound constructor syntax

4.10.2 **New exception class: RecordLockingConflict**

The `RecordLockingConflict` exception class catches record locking conflict errors thrown by file object operations, that is. from methods in the file classes (`Record`, `Recordset`, `SortedRecordset`, `RecordsetCursor`).

An exception is thrown only if there is a Catcher for the exception.

There is no distinction between a Find conflict and a record locking conflict. For example, you can use the following:

```
try
  fd to %foo
  ...
  end find
catch recordLockingConflict
  ...
end try
```

And you can also use:

```
try
  %rec = new(%recnum, exclusive)
  for record %rec
    ...
  end for
catch recordLockingConflict
  ...
end try
```

You can even catch record locking conflicts caused by LoopLockStrength promotions during a loop:

```
%rec = new(%recnum, none, loopLockStrength=share)
...
try for record %rec
  ...
  end for
catch recordLockingConflict
...
end try
```

Or as in the following:

```
%curser = new(%recset, loopLockStrength=none)
...
try
  for record at cursor %curser
    ...
  end for
catch recordLockingConflict
..
end try
```

Or as in this example:

```
fdw01 to %recset = new(loopLockStrength=exclusive)
...
end find
...
try for each record in %recset
  ...
  end for
catch recordLockingConflict
...
end try
```

In the latter example, note that any iteration of the For loop can throw an exception, and there is no way of picking up where you left off (at the next record, say) if you get a RecordLockingConflict exception in such a case. So, unless you always want to lose the

whole loop on a record locking conflict, such a case is almost always better structured with a cursor on the recordset and with Try around the For Record At Cursor loop:

```
fdw01 to %recset = new(loopLockStrength=exclusive)
...
end find
...
%curser = new(%recset, loopLockStrength=exclusive)
repeat while %curser:state eq hasRecord
  try for record at cursor %curser
    ...
  end for
  catch recordLockingConflict
    ...
  end try
  %curser:next
end repeat
```

Note that the For statement can be on the same line as the Try or it can be inside the Try block.

The members of the RecordLockingConflict class are described below. Except for the constructor, *New*, all class members are read-only properties:

UserNumber

The numeric value of the user number (unique per session) of the user that has the conflict.

UserID A longstring that is the *Model 204* userid (login name) of the user that has the conflict.

Filename A longstring that is the name of the file in which the last record locking conflict occurred.

RecordNumber

The numeric value of the *Model 204* internal record number of the record that has the conflict.

New The constructor for the class, *New* lets you set values for each member of the class. Its optional named parameters are the properties of the class:

```
%r1c = New ( [UserNumber = %userNumber]      -
             [, UserID = %userid]           -
             [, Filename = %filename] )     -
             [, RecordNumber = %recordNumber] )
```

The default values of the parameters are the null string or 0, as appropriate.

Example

The following example shows all the properties of the RecordLockingConflict class. While the standard record locking conflict information is available for retrieval via the User Language functions \$rlcfile, \$rlcrec, \$rlcuid, and \$rlcusr, that information is also set in the exception object and retrievable via that object:

```
%rlc is object recordLockingConflict
...
%rec = new(%recnum, none, loopLockStrength=share)
...
try for record %rec
...
end for
catch recordLockingConflict to %rlc
  auditText Conflict on record {%rlc:recordNumber}      -
    in file {%rlc:filename}
  auditText Conflicting user was userid {%rlc:userid}, -
    with user number:{%rlc:userNumber}
end try
```

4.10.3 New exception class: XmlParseError

The XmlParseError exception class catches parsing errors (including syntax, translation, and encoding errors) thrown by the deserialization methods: LoadXml, WebReceive, and the ParseXml method of the HttpResponse class.

The members of the class are described below. Except for the constructor, *New*, all class members are read-only properties:

Reason An enumeration of type XmlParseErrorReason. The possible values are:

SyntaxError A violation of the syntax of an XML document.

InvalidUTF8Encoding

The input UTF8 stream is invalid.

InvalidUTF16Encoding

The input UTF16 stream is invalid.

UntranslatableUnicode

The Unicode input contains a character that is not translatable to EBCDIC. This exception can be avoided using the AllowUntranslatable option of the deserialization method (see [“AllowUntranslatable deserialization option” on page 56](#)).

UntranslatableEBCDIC

The EBCDIC input contains a character that is not translatable to Unicode.

UntranslatableISO

The ISO-8859-n input contains a character that is not translatable to Unicode.

InvalidUnicodeCharacter

The Unicode input contains an invalid character.

The examples below may help to further explain these reasons.

CharacterPosition

The position within the input character stream at or before which the error was detected.

Description

A message that explains the error.

InputHexValue

A hexadecimal string that shows the incorrect input for all reasons except `SyntaxError`.

New

The constructor for the class, `New` lets you set values for each member of the class.

```
%ex = New ( Reason = reasonEnum      -  
           [, CharacterPosition = num] -  
           [, Description = string] ) -  
           [, InputHexValue = string] )
```

The `Reason` argument is required; all other arguments are optional, with default values of the null string or 0 as appropriate.

Example

The following template is used to apply the LoadXml deserialization method to a series of test *input-string* values to demonstrate the XmlParseError exception output:

```

Try
  PrintText {~}: {%d:LoadXml(input-string)}
Catch XmlParseError to %err
  Print ' '
  Print 'Error Reason:' And %err:Reason:ToString
  Print 'Error Description:' And %err:Description
  Print 'Error Character Position:' And
%err:CharacterPosition
  Print 'Error Input Hex Value:' And %err:InputHexValue
  Print ' - - - '
End Try

```

These are the results:

```

%d:LoadXml('<a>'):
Error Reason: SyntaxError
Error Description: XML doc parse error: missing ETag for top
    level element near or before position 4 (end of input)
Error Character Position: 4
Error Input Hex Value:
- - -

%d:LoadXml('<' With 'FF':X With '>')
Error Reason: UntranslatableEBCDIC
Error Description: XML doc parse error: EBCDIC character not
    translatable to Unicode near or before position 3
Error Character Position: 3
Error Input Hex Value: FF
- - -

%d:LoadXml('<a>&#x2122;</a>':U)
Error Reason: UntranslatableUnicode
Error Description: XML doc parse error: invalid Unicode
    character - not translatable to EBCDIC near or
    before position 5
Error Character Position: 5
Error Input Hex Value: 2122
- - -

%d:LoadXml('FEFF003C33':X)
Error Reason: InvalidUTF16Encoding
Error Description: XML doc parse error: expecting additional
    byte of UTF-16 encoding near or before position 2
    (end of input)
Error Character Position: 2
Error Input Hex Value: 33
- - -

```

```
%d:LoadXml('FEFFD800':X)
Error Reason: InvalidUnicodeCharacter
Error Description: XML doc parse error: surrogate point in
                  UTF-16 input near or before position 1
Error Character Position: 1
Error Input Hex Value: D800
- - -
```

```
%d:LoadXml('C181':X)
Error Reason: InvalidUTF8Encoding
Error Description: XML doc parse error: byte 1 of 2 too
                  low in UTF-8 encoding near or before position 2
Error Character Position: 2
Error Input Hex Value: C181
- - -
```

```
%d:LoadXml('FF818256':X)
Error Reason: InvalidUTF8Encoding
Error Description: XML doc parse error: attempt to use
                  4-byte UTF-8 encoding of surrogate point near or
                  before position 2
Error Character Position: 2
Error Input Hex Value: FF
- - -
```

```
%d:LoadXml('EDA080':X)
Error Reason: InvalidUnicodeCharacter
Error Description: XML doc parse error: invalid Unicode
                  character (surrogate range) near or before position 2
Error Character Position: 2
Error Input Hex Value: EDA080
- - -
```

4.11 New utility class: SelectionCriterion

Objects in the SelectionCriterion class are used primarily as input to the collection class searching methods. The searching methods take a SelectionCriterion object as a parameter which provides an item selection specification for the search.

A selection specification, or criterion, is a relational expression that tests a collection item value to determine whether the item (or its item number) is to be returned by the searching method. The SelectionCriterion methods are named for the operation they provide: EQ (equal to), NE (not equal to), LT (less than), LE (less than or equal to), GT (greater than), GE (greater than or equal to). The AND, OR, and NOT methods let you combine comparison operations in a single criterion. The TRUE and FALSE methods respectively match all or no items.

Most SelectionCriterion objects use two parameters to construct a single criterion for selecting a collection item; some use two or more SelectionCriterion objects to construct a single criterion.

As an example specification, the selection criterion `%sel` in the following statement, instantiated by the LT constructor method, matches a number whose square root is less than 10:

```
%sel = lt(squareRoot, 10)
```

Where the left- and right-hand elements of the comparison operation are, respectively, the LT parameter values:

- `squareRoot` is the intrinsic Float method which is applied to each item value before the comparison is made.
- `10` might be any expression of any User Language intrinsic type.

This `%sel` criterion might be used as follows to locate the next ArrayList item after item 3 whose square root is less than 10:

```
%itemnum = %balance:findNextItemNumber(%sel, start=3)
```

The requirements for the SelectionCriterion parameter exemplified by the SquareRoot method above are:

- It must be a method or method variable defined to operate on the type of the items in the collection being searched.
- It must return an intrinsic (number, string, unicode) value.

For more about the method in a SelectionCriterion, see [“Specifying a SelectionCriterion's parameters” on page 44](#).

Additional notes:

- Comparison values are evaluated at SelectionCriterion construction time, not when the search method is evaluated. That is, the following sequence selects items with income greater than or equal to 50000, not 100000:

```
%income = 50000
%sel = ge(income, %income)
%income = 100000
%foo2 = %foo:subsetNew(%sel)
```

- SelectionCriterion have a size limit: the SelectionCriterion, including the values for EQ, NE, LT, LE, GE, and GT comparisons must take less than 252 bytes.

ANDs and ORs take 4 bytes, and comparison selectors have 12 bytes of overhead plus the length of the value rounded to a 4-byte multiple. So, for example, an And condition with seven 20-byte values would take $4+7*(20+12)$, or 228, bytes.

You can work around a criterion that exceeds 252 bytes by encapsulating some or all of the conditions in a local method, since a local enhancement method can be the method parameter for a comparison SelectionCriterion.

4.11.1 Declaring a SelectionCriterion object variable

The SelectionCriterion class operates on specific objects, so a variable of the SelectionCriterion class must be qualified with the object to which it applies: For example:

```
%sel1 is selectionCriterion for object myClass
%sel2 is selectionCriterion for object xmlNode
%sel3 is selectionCriterion for longstring
```

The declaration for `%sel` in the example in the preceding section might be:

```
%sel is selectionCriterion for float
```

In general, the syntax for declaring a SelectionCriterion object variable is:

```
<objvar> Is Object SelectionCriterion For <itemtype>
```

Where:

<objvar> The name of the SelectionCriterion object variable.

<itemtype> The datatype of the items in the collection to be searched.

4.11.2 Specifying a SelectionCriterion's parameters

As stated earlier, most of the SelectionCriterion methods accept two parameters, the first a “method value” and the second an intrinsic value. Regarding the first, just as a User Language “numeric” argument can be a numeric literal or a variable or even an expression that results in a number, a method value argument can be a method literal or a method variable or an expression that results in a method.

More specifically, a method value is a value assigned to a method variable. And for a SelectionCriterion, the method variable's *implicit* declaration is:

```
Is Function (<itemtype>):<methname> Is <intrinType>
```

Where:

- <itemtype>** The class of the items in the collection to be sorted.
- <methname>** A method name, but merely a placeholder in an actual declaration. Any method (system or user), class Variable or Property, local method, or method variable that fits the declaration can be used in the SelectionCriterion.
- <intrinType>** A User Language intrinsic class type returned by the function.

The method value argument must conform to the following rules:

- It must return an intrinsic value.
- It cannot have any parameters other than the method object.

The following examples show different method argument types as the first parameter in a SelectionCriterion. In the examples below, the method values are respectively a class Variable, a local method, and a special method value, all of which satisfy the Function template described above.

1. The selectionCriterion in the following example uses a class Variable, `Income`, as the criterion method parameter. The collection method, `SubsetNew`, returns a `NamedArraylist` that contains all the items that satisfied the selection criterion.

```
Begin
class foo
  public
    variable name is string len 20
    variable status is string len 8
    variable income is float
    constructor newf(%a is string len 10, -
                    %b string len 8, %c is float)
    function myprint is longstring
  end public

  constructor newf(%a is string len 10, -
                  %b string len 8, %c is float)
    %this:name = %a
    %this:status = %b
    %this:income = %c
  end constructor

  function myprint is longstring
    return %this:name with ' ' with -
          %this:status with ' ' with -
          '(income: ' with %this:income with ')'
  end function
end class

%fool is namedArraylist of object foo
%fool = new
%fool('Jim') = newf('Red', 'employed', 45000)
```

```
%fool('Jed') = newf('Black', 'retired', 30000)
%fool('Jan') = newf('Green', 'employed', 55000)
%fool('Jud') = newf('Brown', 'expired', 0)
%fool('Jon') = newf('White', 'tired', 70000)

%fool = %fool:subsetNew(gt(income, 0))

for %i from 1 to %fool:count
    print %fool:namebynumber(%i) And -
          %fool:itembyNumber(%i):myprint
end for

End
```

The result is:

```
Jan Green employed (income: 55000)
Jed Black retired (income: 30000)
Jim Red employed (income: 45000)
Jon White tired (income: 70000)
```

2. The second parameter in most SelectionCriterion methods is a User Language intrinsic-type value, often a string or numeric literal. Consequently, you cannot specify a criterion that compares two method values. For example, for a `Cust` class (similar to the class in the previous example) with variables `Expenses` and `Income`, you might want to find all the customers with expenses greater than income — but the following **is not allowed**:

```
%custDebtors = %custs:subsetNew(gt(expenses, income))
```

However, you can create a local method with which you can achieve the same result:

```
local function (cust):netIncome is float
    return %this:income - %this:expenses
end function
%custDebtors = %custs:subsetNew(lt(netIncome, 0))
```

3. The SelectionCriterion in the following request uses the special method value `This` as the method parameter, and the `And` constructor combines two criteria to create one criterion. Valid only when applied to User Language intrinsic method objects, `This` simply returns the value of the method object.

```
Begin
%fooflt is arraylist of float
%fooflt = list(111, 29, 93, 77, -345)

%sel is object selectioncriterion for float
%sel = and(gt(this, 50), le(this, 93))
%fooflt = %fooflt:subsetnew(%sel)
```

```
%fooflt:print
End
```

The result is:

```
1: 93
2: 77
```

4.11.3 The EQ, NE, GE, GT, LE, and LT constructors

These shared methods each create a new SelectionCriterion object that is a relational expression used to select items from a collection. Each of these constructors provides a different comparison operator. The EQ method, for example, constructs an equality expression that selects a collection item if the expression is true for that item. The other methods construct, respectively, not-equal-to, greater-than-or-equal-to, greater-than, less-than-or-equal-to, or less-than expressions.

The collection searching method that makes use of a selection criterion specifies:

- Whether to return the first item or item number or all items that satisfy the selection criterion.
- Where in the collection to begin searching.

The first of the two parameters in a selection criterion specifies a function that is applied to a collection item. The function result is the “left-side” operand in the criterion's expression. The second selection criterion parameter is a string or numeric that is the “right-side” operand in the expression. `EQ(name, 'Ortiz')` specifies the expression `name='Ortiz'`.

The function parameter of one of these constructors might simply be an identity function that returns the item value. Or, for example, it might be a function that returns the value of a class member for an item that is an object. This function must be a method that operates on the item type and returns a User Language intrinsic datatype (Float, String, Longstring, or Unicode) value.

```
%selc = [% (selectionCriterion for itemtype):] -
        EQ(function, value)
```

EQ syntax (same for NE, GE, GT, LE, and LT)

Notes:

- For more information about the *function* parameter, see [“Specifying a SelectionCriterion's parameters” on page 44](#).
- The *function* parameter is a method value, not a User Language expression, and you may not specify a function that itself has an argument. The SelectionCriterion syntax does not provide for specifying a parameter for the *function* parameter. If

necessary, the workaround for this restriction is to define a local method that accepts an argument, then use that method as the *function* parameter.

- The *function* may be `This`, an identity method that is valid for User Language intrinsic method objects only. The `This` method returns the value of the item to which it is applied. See example “3.” on page 46.

Examples:

1. The following criterion matches a number greater than 10:

```
%sel = gt(this, 10)
```

The following matches a number with an absolute value less than or equal to 100:

```
%sel = le(absolute, 100)
```

The following matches numbers less than 70 and greater than or equal to 95:

```
%sel = or(lt(this,70), ge(this,95))
```

2. The following request fragment selects two items from the `%foo1` NamedArraylist in example “1.” on page 45:

```
...
%sel is object selectionCriterion for object foo
%sel = EQ(status, 'Employed')
%foo1 = %foo1:subsetNew(%sel)

for %i from 1 to %foo1:count
  print %foo1:namebyNumber(%i) And -
        %foo1:itembyNumber(%i):myprint
end for
...
```

The result is:

```
Jan Green employed (income: 55000)
Jim Red employed (income: 45000)
```

4.11.4 The OR and AND constructors

These shared methods each create a new SelectionCriterion object that is an expression used to select the items in a collection. Each constructor uses a different logical operator to form an expression that combines one or more SelectionCriterion objects.

An OR criterion returns true for a collection item if any of the component SelectionCriterion expressions are true for the item; otherwise it returns false. An AND criterion returns true if all of the component SelectionCriterion expressions are true for the item; otherwise it returns false.

```
%selc = [% (selectionCriterion for itemtype):] -
        OR(criterion1 [, criterion2] ... [, criterionN])
```

OR syntax (same for AND)

All OR and AND SelectionCriterion conditions are short-circuiting conditions. That is, if any of the conditions in an OR return True, the subsequent conditions are not evaluated and the OR returns True. Similarly, if any of the conditions in an AND return False, the subsequent conditions are not evaluated, and the AND returns a False.

Therefore, it is wise to put the most likely conditions first in an OR, and it is wise to put the least likely first in an AND. For a mix of conditions where some are simply variable references and others require method evaluation, it probably is best to put the variable references first, as these are probably much cheaper to evaluate.

Examples:

1. The following criterion matches numbers less than 70 and greater than or equal to 95:

```
%sel = OR(lt(this,70), ge(this,95))
```

2. The following criterion matches numbers that are less than or equal to 33 but not 30:

```
%sel = AND(le(this, 33), ne(this,30))
```

4.11.5 The NOT constructor

This shared method creates a new SelectionCriterion object that is a logical negation of its SelectionCriterion parameter. A NOT criterion returns true for a collection item if NOT's component SelectionCriterion expression is false for the item; otherwise it returns true.

```
%selc = [% (selectionCriterion for itemtype):] -  
        NOT(criterion)
```

NOT syntax

The NOT constructor is **never** necessary, and anything you can do with the NOT can be done (probably more clearly) otherwise. For example, the following two criteria are identical:

```
%sel = not(gt(this,90))  
  
%sel = le(this,90))
```

And the following two criteria are identical:

```
%sel = not(or(lt(this,70), ge(this,95)))  
  
%sel = and(ge(this,70), lt(this,95))
```

And these criteria are identical:

```
%sel = not(true)  
  
%sel = false
```

Internally, NOTs are always converted to the inverse of the parameter criterion.

4.11.6 The TRUE and FALSE constructors

These shared methods take no parameters and create a new SelectionCriterion object. A TRUE criterion returns true for any collection item; A FALSE criterion returns false for any collection item.

TRUE and FALSE thus provide the equivalent of an empty SelectionCriterion, and they substitute for a New constructor in the SelectionCriterion class.

```
%selc = [% (selectionCriterion for itemtype):]TRUE  
  
%selc = [% (selectionCriterion for itemtype):]FALSE
```

TRUE and FALSE syntax

Notes:

- TRUE and FALSE exist to simplify dynamic SelectionCriterion applications. If you are dynamically generating a SelectionCriterion, under some conditions you may want the criterion to select all objects or to select no objects.

In addition, if you are building a SelectionCriterion by using OR ([“The OR and AND constructors” on page 49](#)) in conditions, it might be useful to start out with a FALSE SelectionCriterion:

```
%sel = FALSE
if <somecondition1> then %sel = OR(%sel, EQ(foo, 'A'))
end if
if <somecondition2> then %sel = OR(%sel, EQ(foo, 'B'))
end if
if <somecondition3> then %sel = OR(%sel, EQ(foo, 'C'))
end if
```

Similarly, if building a SelectionCriterion by using AND in conditions, it might be useful to start out with a TRUE SelectionCriterion.

- These SelectionCriterion methods are constructors and as such can be called with no method object, with an explicit class name, or with an object variable, even if that object is null:

```
%selc1 = TRUE

%selc2 = %(selectionCriterion for float):FALSE

%selc3 = %selc3:TRUE
```

- The following statements are equivalent:

```
%sel = and(ge(this, 22), true)

%sel = ge(this, 22)
```


The following sections describe changes in the *Janus SOAP Xml API* in this release.

5.1 XmlDocs now maintained in Unicode

As of *Sirius Mods* version 7.6, XmlDocs contain Unicode rather than EBCDIC; this is true for all string values, names, prefixes, and URIs. As a consequence, most of the arguments and results of the Xml API methods that formerly were strings are now Unicode strings.

This change will allow you to store Unicode in an XmlDoc, thus achieving the W3C XML Recommendation (in which characters are Unicode), but without needing to change your existing Xml API applications. This is because automatic conversions are done between EBCDIC and Unicode in *Janus SOAP*.

For example, you can still code:

```
%d:AddElement('name', 'value')
```

The EBCDIC character strings above are automatically converted from EBCDIC to Unicode. Similarly, you can code:

```
%str = %n:Value
```

If the variable `%str` above is declared as type `String` or `Longstring`, then the Unicode result of the `Value` method is automatically converted from Unicode to EBCDIC when it is stored in `%str`.

This feature did require some changes to existing *Janus SOAP Xml API* methods. Those changed methods are included in the following sections, even where the change is only of an argument or result to Unicode.

5.2 New methods

5.2.1 AllowNull XmlDoc property

A new XmlDoc property, `AllowNull`, is now available. This Boolean property, which defaults to `False`, may be set to `True`, which allows the XmlDoc to contain null characters (U+0000) in node values.

Once this property has been set to True, null characters are allowed for deserialization into the XmlDocument. For example:

```
<a>Element contains null: &#0;</a>
```

Null characters are also allowed for “direct setting” of a node's value. For example:

```
%nod:AddElement('a', 'Element contains null:' With $X2C('00'))
```

When nodes have been added to an XmlDocument that has `AllowNull=True`, a “subtree copy” operation (that is, either `AddSubtree` or `InsertSubtreeBefore`) is not allowed using it as the source (argument 1) if the target (method object) of the operation is an XmlDocument with `AllowNull=False`.

Notes:

- All Unicode characters except null can always be stored in an XmlDocument, so when `AllowNull` is set to True, all Unicode characters can be stored. As mentioned in [“InvalidChar XmlDocument property now obsolete” on page 69](#), if you had been using the now obsolete `InvalidChar=Allow` setting to allow null characters in an XmlDocument, this purpose can be achieved via `AllowNull=True`.
- Deserialization of Unicode characters that do not translate to EBCDIC is not allowed unless the `AllowUntranslatable` deserialization option ([“AllowUntranslatable deserialization option” on page 56](#)) is used.
- When providing EBCDIC characters to be stored in an XmlDocument, those EBCDIC characters must be translatable to Unicode. If you have EBCDIC strings which may not be translatable, you must handle those before passing them to an XmlDocument update operation. For example, you may be able to use the `Untranslatable` argument of the `EbcdicToUnicode` function.
- A null character in an XmlDocument is serialized as `�`.

5.2.2 AppendValue method in XmlNode class

A new XmlNode subroutine, `AppendValue`, appends to the value of a node in an XmlDocument.

```
XmlNode:AppendValue(stringListOrUnicodeString)
```

AppendValue syntax

Where:

- The method object is an XmlNode other than the Root node. If it is an Element node, it may have at most one child, and any such child must be a Text node.

- The method argument is either a Unicode string or a Stringlist object:
 - If a Unicode string, the value of the node is set to its current value followed by the Unicode string.
 - If a Stringlist object, the value of the node is set to its current value, followed by the Stringlist items converted from EBCDIC to Unicode, with Unicode linefeed characters between the Stringlist items.

For example, the following fragment appends a Unicode string and then a Stringlist to an Element value:

```
%n = %x:AddElement('a', 'begin value')
%n:AppendValue('_some more_')
Text To %s1
    and more
    and that's all
End Text
%n:AppendValue(%s1)
%n:Print
```

The result is:

```
<a>begin value_some more_and more&#xA;and that's all</a>
```

5.2.3 NoEmptyElement property of element XmlNodeNodes

The new XmlNode property, `NoEmptyElement`, may be specified for any Element node. If set for an Element that contains no children, the Element is serialized with a separate start tag and end tag, rather than with a single empty element tag. For example, `<address></address>` versus `<address/>`.

This formatting option can be useful if you are using the *Janus SOAP ULI* to generate XHTML. Tests show that some browsers work correctly for certain childless elements only if they have an empty element tag, and for other childless elements they work correctly only if there are separate start and end tags. (Therefore, you cannot obtain a “blanket” suppression of empty element tags via using the `NoEmptyElement` option of the serialization methods.)

The `NoEmptyElement` property accepts and returns a Boolean value. Setting it to `True` suppresses an empty element tag. The default is `False`.

For example:

```
%texta = %form:addElement('textArea')
%texta:NoEmptyElement = True
%texta:AddAttribute('name', 'foo')
%texta:AddAttribute('rows', '15')
%texta:AddAttribute('cols', '40')
%texta:Print
```

The result is:

```
<textArea name="foo" rows="15" cols="40"></textArea>
```

5.3 Changes to existing methods

5.3.1 AllowUntranslatable deserialization option

A new option, `AllowUntranslatable`, is now available for the `LoadXml` and `WebReceive` methods, as well as for the `ParseXml` method of the `HttpResponse` class. When this option is specified, all valid Unicode strings are allowed in the XML document. When this option is not specified, Unicode strings that are not translatable to EBCDIC are not allowed.

How you use this option depends upon the application's handling of the `XmlDoc` into which you are deserializing. The basic rule is to use `AllowUntranslatable` only if the application checks for translatability when accessing parts of the `XmlDoc` that may have untranslatable Unicode content.

- `AllowUntranslatable` **not** specified

In this case, any access to the deserialized content is performed without any Unicode to EBCDIC translation errors. This approach detects (and throws an `XmlParseError` exception with reason `UntranslatableUnicode` for) any untranslatable Unicode in the serialized input XML document. The `XmlParseError` exception is explained in [“New exception class: XmlParseError” on page 39](#).

That is to say, given the following fragment:

```
%doc:WebReceive
...
%val Longstring
%val = %doc:Value(%xpath)
```

The assignment to the EBCDIC string `%val` cannot not fail due to a Unicode translation problem: if there is any untranslatable Unicode (including, of course, strings in the XML document which your application never accesses), the `WebReceive` operation fails.

- AllowUntranslatable specified

In this case, all Unicode characters in the serialized input XML document are allowed and stored in the XmlDocument. However, if the application later accesses content that is not translatable to EBCDIC, and the access performs Unicode-to-EBCDIC translation which would cancel the request, the request gets cancelled.

The code below shows a way to get the benefit of specifying AllowUntranslatable while limiting the risk of request cancellation. In the example, it is believed that only the element `comments` might contain untranslatable Unicode among all the data accessed from the XML document:

```
%resp:ParseXml(%doc, 'AllowUntranslatable')
...
%uVal Unicode
%val Longstring
%uVal = %node:Value('comments')
Try %val = %uVal:UnicodeToEbcDic
Catch CharacterTranslationException
    %val = %uVal:UnicodeToEbcDic(CharacterEncode=True)
    Print 'Untranslatable Unicode, character encoded:' -
        And %val
End Try
```

Note: Unicode values, untranslatable or not, are always allowed when they are added to an XmlDocument using one of the methods which “directly store” into an XmlDocument. For example, the following fragment adds an Element node with a value which is the Unicode trademark sign:

```
%node:AddElement('notation', '&#x2122;':U)
```

5.3.2 CharacterEncodeAll option of Print, Audit, and Trace

A new option, `CharacterEncodeAll`, is available for the Print, Audit, and Trace methods. This option uses character encoding in all contexts to display Unicode characters that do not translate to EBCDIC.

With or without this option, non-translatable Unicode characters in Attribute or Element values are displayed as character references. For example:

```
%doc:AddElement('top', '&#x2122;':U)
%doc:Print
```

The result of this fragment is:

```
<top>&#x2122;</top>
```

However, with default serialization options, when an untranslatable Unicode character occurs in a context other than element or attribute value (that is, a name, comment, or PI), character encoding is *not* used. For example, the following statements result in a request cancellation:

```
%doc:AddElement('&#x2122;':U)
%doc:Print
```

The Print method fails, attempting to translate the element name, the U+2122 character, to EBCDIC. This request cancellation can be overcome using CharacterEncodeAll:

```
%doc:AddElement('&#x2122;':U)
%doc:Print(, 'CharacterEncodeAll')
```

The result of the above fragment is:

```
<&#x2122;/>
```

Note: The result of a Print with CharacterEncodeAll can be misleading. Request cancellation is avoided, but it produces multiple EBCDIC characters where only a single Unicode character is stored.

The XmlDocument, %doc, above is not a legal XML document, because the ampersand (&) is not a legal name character. Similarly, for an untranslatable Unicode character added to a document with AddComment or AddPI: printing with CharacterEncodeAll produces a stream of characters that informs about a single character reference but, if deserialized, would result in multiple stored characters. The standard XML syntax does not recognize character references as such in names, Comments, and PIs.

5.3.3 InvalidChar property removed

The InvalidChar property of the XmlDocument class is no longer supported. Requests containing InvalidChar will fail under *Sirius Mods* version 7.6. If you are using InvalidChar in source code to allow null characters, you can use the AllowNull property (“AllowNull XmlDocument property” on page 53) instead.

5.3.4 InvalidCharacterPosition shared XmlDocument function

The InvalidCharacterPosition method is enhanced as follows:

- It now accepts an optional, name-required argument: `AllowNull`. AllowNull takes a Boolean value. If AllowNull is `True`, an input null character is **not** considered invalid.
- It now validates **either** an EBCDIC or Unicode string. Formerly, only an EBCDIC argument was allowed.

InvalidCharacterPosition returns zero if its argument string is:

- Unicode, and all characters are legal in an XML document (that is, it does not contain any null characters).
- EBCDIC, and all characters are translatable to Unicode characters other than the null character.

Otherwise, it returns the character position of the first character that does not meet these criteria.

5.3.5 IsValidString shared XmlDocument function

The IsValidString method now accepts **either** an EBCDIC or Unicode string. Formerly, only an EBCDIC argument was allowed. IsValidString now returns **True** if its argument string is:

- Unicode, and all characters are legal in an XML document (that is, it does not contain any null characters).
- EBCDIC, and all characters are translatable to Unicode characters other than the null character.

Otherwise, it returns **False**.

Note: IsValidString is deprecated, in favor of the InvalidCharacterPosition function.

The following features are new or changed in *Janus TCP/IP Base*.

6.1 New forms of JANUS LOADXT

There are two new forms of the JANUS LOADXT command:

```
JANUS LOADXT xtabName UNICODE
JANUS LOADXT xtabName DEFAULT
```

JANUS LOADXT xtabName UNICODE

This creates a Janus translation table which can be referenced, for example, in the `XTAB` parameter of the JANUS DEFINE or JANUS WEB command.

The name of the table is given in the third word of the command, shown as ***xtabName*** above.

The translations defined in the table are the same as the translations between ASCII and EBCDIC defined in the current Unicode table, except that, since there is no concept of “untranslatable” in the use of the Janus translation tables, the following translations are used:

ASCII to EBCDIC Untranslatable ASCII code points are translated to EBCDIC X'FF'.

EBCDIC to ASCII Untranslatable EBCDIC code points are translated to ASCII X'3A' (the ASCII colon character - “:”).

The translations defined between ASCII and EBCDIC are the translations used by the AsciiToEBCDIC and EBCDICToAscii methods.

JANUS LOADXT xtabName DEFAULT

This command will set the translation tables for the designated ***xtabName*** to the default (i.e., in the absence of any LOADXT commands) Janus tables.

This can be used as an “undo” command; for example, if you had issued:

```
JANUS LOADXT STANDARD UNICODE
```

Then the following command will revert the STANDARD xtab to its default:

```
JANUS LOADXT STANDARD DEFAULT
```

CHAPTER 7 *Janus Sockets*

The following features are new or changed in *Janus Sockets*:

7.1 SMTP Helper

7.1.1 New parameters for AddPart method

The new `Disposition` and `Description` parameter options are added to the SMTP Helper `AddPart` method. These name required parameters are strings that provide content-disposition and content-description headers for attachments added with `AddPart`.

`AddPart` adds a content-disposition header or a content-description header to an e-mail attachment only if you specify, respectively, a `Disposition` or a `Description` value.

Sir2000 Field Migration Facility

The following are new or changed features in the *Sir2000 Field Migration Facility*:

8.1 SIRFIELD command not allowed with FILEORG=X'100'

The *Sir2000 Field Migration Facility* cannot be used in a file that has `FILEORG=X'100'`. It is an error to issue the `SIRFIELD` command with such a file.

This chapter lists any compatibility issues with prior versions of the *Sirius Mods* and any bugs which have been fixed in this version of the *Sirius Mods* but had not, as of the date of this release, been fixed in the previous generally available version (7.5).

In general, backward incompatibility means that an operation which was previously performed without any indication of error, now operates, given the same inputs and conditions, in a different manner. We may not list as backwards incompatibilities those cases in which the previous behaviour, although not indicating an error, was “clearly and obviously” incorrect, and which are introduced as normal bug fixes (whether or not they had been fixed with previous maintenance).

9.1 Backwards incompatibilities

Backwards incompatibilities are described per product in the following sections.

9.1.1 Janus SOAP ULI

9.1.1.1 ToXmlDoc default changed to AttributeNames=True

The default value of the `AttributeNames` argument of the `ToXmlDoc` method in the `Record` class has been changed.

Formerly, the default was `False`; the default now is `True`.

The reason for this change is that the `AttributeNames=True` format is more well suited to operations on the `XmlDoc`, particularly a record copying operation.

This change was also introduced as part of maintenance in the 7.3, 7.4, and 7.5 *Sirius Mods*.

9.1.1.2 Recordset LockStrength inconsistencies

Two similar fixed problems may introduce backwards incompatibilities:

- Formerly, if a `FD` or `FDWOL` statement found no records, the target recordset `LockStrength` would be incorrectly set to `None` (when they should have been set according to their `Find` statement types to, respectively, `Share` and `None`). This

made it possible for a user to do an FD followed by an AddRecordset, and the resultant recordset might be locked None or Share depending on whether the FD found any records.

As of *Sirius Mods* version 7.6, the LockStrength of the target Recordset of a FD statement that finds no records is always set to **Share**.

- The AddRecordsetNew, RemoveRecordsetNew, and AndRecordsetNew methods are supposed to set the LockStrength and LoopLockStrength of the their output Recordset objects to be the same as that of their method objects. But in fact these methods always produced unlocked recordsets.

As of *Sirius Mods* version 7.6, these methods now correctly set these output Recordset locking strengths to be the same as that of the method objects.

9.1.1.3 Print/Audit/Trace statements encode Unicode ampersands

As described in “[Print/Audit/Trace statements character-encode Unicode values](#)” on [page 7](#), the Print, Audit, and Trace statements now encode Unicode data for display. In the case of the ampersand, this can change the result. For example, in version 7.5 of the *Sirius Mods*, the following fragment:

```
%u Unicode Initial('Jack & Jill')  
Print %u
```

produced the following result:

```
Jack & Jill
```

In version 7.6, however, it produces the following result:

```
Jack & Jll
```

9.1.2 Janus SOAP Xml API

The following backwards compatibility issues have been introduced in the *Janus SOAP* Xml API.

9.1.2.1 Serialization of attributes always bracketed by quotes

In version 7.5 of *Janus SOAP*, some attempt is made to use the apostrophe (') to bracket the serialization of an attribute if it contains a quote ("). In version 7.6, attributes are always bracketed by quotes.

For example, consider the following fragment:

```
Begin
%d:AddElement('a'):AddAttribute('b', '"Wow"')
%d:Print
```

This produces the following result with version 7.5:

```
<a b=' "Wow" ' />
```

However, starting with version 7.6, it produces the following result:

```
<a b="&quot;Wow&quot;" />
```

9.1.2.2 **InvalidChar XmlDoc property now obsolete**

In versions 7.3 and 7.5 of *Janus SOAP*, although you are allowed to set and retrieve the `InvalidChar` property of an `XmlDoc` object, this property is ignored and has no effect. In versions 7.2 and older of *Janus SOAP*, this property allowed certain EBCDIC characters (nulls, for example) to be stored in an `XmlDoc` which otherwise were not allowed.

Starting with version 7.6 of *Janus SOAP*, all Unicode characters may be stored in an `XmlDoc`, except for the null character (U+0000), which is prohibited by the XML Recommendation.

Therefore, the `InvalidChar` property would be extremely misleading, so it has now been removed from *Janus SOAP*.

If you had been using `InvalidChar=Allow` to allow null characters in an `XmlDoc`, this purpose can be achieved via `AllowNull=True`, described in [“AllowNull XmlDoc property” on page 53](#).

Also note that when providing EBCDIC characters to be stored in an `XmlDoc`, those EBCDIC characters must be translatable to Unicode. If you have EBCDIC strings which may not be translatable, you must handle those before passing them to an `XmlDoc` update operation; for example, you may be able to use the `Untranslatable` argument of the `EbcdicToUnicode` function.

9.1.2.3 **Change to structure of LoadParameterInfo document**

Now the `LoadParameterInfo` method builds an `XmlDoc` such that:

- The name of each *Model 204* parameter is used as the value of the 'name' attribute of a 'parameter' element in the `XmlDoc`; formerly, the name of that element was the name of the parameter.

- The description of each *Model 204* parameter is provided as the value of the 'description' element child of a 'parameter' element in the XmlDoc; formerly, the description was provided as the value of a text child of the element whose name was the name of the parameter.

9.2 Fixes in Sirius Mods 7.6 but not in 7.5

This section lists fixes to functionality existing in the *Sirius Mods* version 7.5 but which, due to the absence of customer problems, have not, as of the date of the release, been fixed in that version.

- There are no fixes in *Sirius Mods* 7.6 that are not available in previous versions.

9.3 Version corequisites

This section lists any restrictions on usage of various products (including *Sirius Mods* itself) that will be imposed by use of version 7.6 of *Sirius Mods*.

- There are no corequisites associated with *Sirius Mods* 7.6.