

---

# Notes for Sirius Mods Release 7.2

**February, 2008**

---



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](mailto:support@sirius-software.com)  
World Wide Web: <http://sirius-software.com>

November 28, 2008

© 2008 Sirius Software, Inc.

---

## *Proprietary Notices*

The following products:

- *Janus SOAP*
- *Janus Sockets*
- *Janus Web Server*
- *Janus Debugger*
- *Sirius Debugger*
- *Sirius Functions*
- *SirFact*

are proprietary products of Sirius Software, Inc.:

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

**Model 204™** is a proprietary product of Computer Corporation of America:

**Computer Corporation of America**  
**200 West St.**  
**3rd Floor West**  
**Waltham, MA 02451**  
**USA**

---

## Contents

<b>Proprietary Notices</b> . . . . .	<b>ii</b>
<b>Contents</b> . . . . .	<b>iii</b>
<b>Chapter 1: Introduction</b> . . . . .	<b>1</b>
<b>Chapter 2: Maintenance and Support</b> . . . . .	<b>3</b>
Model 204 support . . . . .	3
<b>Chapter 3: All or Multiple Products</b> . . . . .	<b>5</b>
Html/Text statement enhancements . . . . .	5
To Audit, To Print, and To Trace options . . . . .	5
Audit, Print, and Trace keywords . . . . .	5
AuditText, PrintText, and TraceText options . . . . .	6
LiteralsToTemp keyword . . . . .	6
{~} directive . . . . .	6
Longstrings in images . . . . .	7
Report writer parameters . . . . .	7
Janus stream logging . . . . .	7
SirScan accepts large datasets . . . . .	8
<b>Chapter 4: Janus Debugger and Sirius Debugger</b> . . . . .	<b>9</b>
Features available only with Sirius Mods 7.2 . . . . .	9
Jump to arbitrary statement . . . . .	9
Audit Trail display for procedures excluded by White List processing . . . . .	9
Interrupting White List or Run Until processing . . . . .	10
New features for Sirius Mods 6.9 through 7.2 . . . . .	10
Enhanced support for macros . . . . .	10
New macro-only commands . . . . .	10
Passing a command argument to a macro . . . . .	11
A console for macro information . . . . .	11
Macro command line . . . . .	11
Macro-running indicator . . . . .	11
Watching mixed-case global variables . . . . .	12
Keyboard-invoked consecutive searches . . . . .	12
New mappable commands . . . . .	12
ON UNIT code debugging . . . . .	13
Reporting for DEBUGGERSERVER and DEBUGGERCLIENT Janus ports . . . . .	13

<b>Chapter 5: Janus SOAP ULI</b> . . . . .	<b>15</b>
The Dataset class . . . . .	15
Dataset example . . . . .	16
RecordFormat Enumeration . . . . .	16
The DatasetState Enumeration . . . . .	17
Dataset methods . . . . .	17
Intrinsic classes . . . . .	18
Intrinsic method objects . . . . .	20
Intrinsic method syntactic oddities . . . . .	22
Intrinsic Float methods . . . . .	24
SquareRoot function . . . . .	25
Intrinsic String methods . . . . .	25
Base64toString function . . . . .	25
HexToString function . . . . .	25
Left function . . . . .	26
Length function . . . . .	26
PositionIn and PositionOf functions . . . . .	26
RegexMatch function . . . . .	26
RegexReplace function . . . . .	27
RegexSplit function . . . . .	27
Right function . . . . .	28
StringToBase64 function . . . . .	28
StringToHex function . . . . .	28
Substring function . . . . .	28
Exception handling . . . . .	28
Exception class definitions . . . . .	30
Throwing exceptions . . . . .	31
Specifying a Throws clause . . . . .	32
Using the Throw statement . . . . .	32
Exception classes extending other exception classes . . . . .	34
Try and Catch . . . . .	34
Some differences with other languages . . . . .	36
Nesting Try/Catch blocks . . . . .	36
OnThrow and OnUncaught . . . . .	37
System Exception classes . . . . .	38
DaemonLost class . . . . .	39
InvalidBase64Data class . . . . .	39
InvalidHexData class . . . . .	40
InvalidRegex class . . . . .	41
NoFreeDaemons class . . . . .	42
Enhancement methods . . . . .	42
Enhancement methods and inheritance . . . . .	45
Intrinsic Enhancement methods . . . . .	45
Enhancement methods for Collections . . . . .	47
ReadOnly and Protected variables . . . . .	48
New System class methods . . . . .	49
The Arguments method . . . . .	49

The CallStack method . . . . .	50
The CallStackCaller method . . . . .	50
The CallStackDepth method . . . . .	50
<b>Chapter 6: Janus SOAP Xml API . . . . .</b>	<b>51</b>
Multiple XPath predicates . . . . .	51
not() function in XPath predicates . . . . .	51
Nested XPath predicates . . . . .	52
Boolean parentheses in XPath predicates . . . . .	52
URI now allows hexadecimal escape (%hh) . . . . .	52
<b>Chapter 7: Janus Sockets . . . . .</b>	<b>55</b>
HTTPRequest class . . . . .	55
Send method . . . . .	55
<b>Chapter 8: Janus Web Server . . . . .</b>	<b>57</b>
Support for additional HTTP method types . . . . .	57
<b>Chapter 9: Sirius Functions . . . . .</b>	<b>59</b>
NOURGMSG option of \$Wakeup . . . . .	59
\$Lstr_Word return word longer than 255 bytes . . . . .	59
<b>Chapter 10: Compatibility/Bug fixes . . . . .</b>	<b>61</b>
Backwards incompatibilities . . . . .	61
Janus SOAP Xml API . . . . .	61
Additional URI restrictions . . . . .	61
Janus Web Server . . . . .	61
Support for additional HTTP method types in Janus Web rules . . . . .	62
Fixes in Sirius Mods 7.2 but not in 7.1 . . . . .	62
Various URIs now recognized to be invalid . . . . .	62
Processing certain alternatives in Regex expressions . . . . .	63
Version corequisites . . . . .	64



---

**CHAPTER 1** ***Introduction***

This document lists the enhancements and other changes contained in *Sirius Mods* version 7.2, which was released in February, 2008. The previous version of the *Sirius Mods*, 7.1, was available in July, 2007.



CHAPTER 2 *Maintenance and Support*

**2.1 Model 204 support**

*Sirius Mods* version 7.2 supports only *Model 204* version V6R1.



## *All or Multiple Products*

### 3.1 **Html/Text statement enhancements**

The User Language Html and Text statements, alternatives to a series of PRINT statements, are enhanced as described in the following subsections.

#### 3.1.1 **To Audit, To Print, and To Trace options**

These three new Html/Text statement To options let you direct the contents of an Html/Text block or line to the destinations associated with the User Language statements Audit, Print, or Trace. Respectively these destinations are the journal/audit trail, the current print output device, or the trace destination (which may be the audit trail, print device, or CCATEMP trace table, or a combination of them).

For example, the following statement directs the Text block content to the audit trail:

```
text to audit
  Now ({$time}) is the time for all good men
  to come to the aid of their company.
end text
```

To direct a single line of output, you can use the Data keyword as in the following:

```
text to audit data It ({$time}) was the best of times
```

The next feature item describes an alternative to the single-line format in the example above.

#### 3.1.2 **Audit, Print, and Trace keywords**

These new keywords act like the Data keyword: they denote that the HTML block consists of only the text that follows the keyword on the logical line. Further, they identify the output destination of the line's text, implying To Audit, To Print, and To Trace, respectively.

The following example of the Audit keyword shows that the new keyword simplifies `Text To Audit Data` (from the second example in [“To Audit, To Print, and To Trace options”](#)) to `Text Audit`:

```
text audit It ({$time}) was the best of times
```

This format is likely preferable to the new block format option (in “[To Audit, To Print, and To Trace options](#)” on page 5) if you need to specify some Text statement options, as in the following example:

```
text noexpr audit The set is {2, 3, 5, 8, 13, 21, ...}
```

### **3.1.3 AuditText, PrintText, and TraceText options**

These options are simply shortened forms of Text Audit, Text Print, and Text Trace, respectively, that place the action first. These are likely to be the common way of auditing, printing, or tracing text when no additional Text statement options need to be specified.

An AuditText example follows:

```
auditText It ({$time}) was the best of times
```

### **3.1.4 LiteralsToTemp keyword**

When this keyword is specified in a Text statement, the literal data in the statement is stored in CCATEMP rather than STBL, avoiding excessive demands on STBL space. Designed for a Text statement with an unusually large amount of literal data, this is an alternative to setting the X'01' bit of the SIRCOMP user parameter (which performs the same function).

### **3.1.5 {~} directive**

In a Text/Html statement block, content enclosed in expression start and end characters is evaluated as if it were the right-hand side of a User Language assignment, and the resulting value takes the place of the expression (and expression start and end characters) in the HTML block. For example, `printText {$date}` produces `08-02-29`.

The special expression `{~}`, however, directs the compiler to replace this expression with the literal character content of the next expression appearing on the same line. So `printText {~} {$date}` produces `$date 08-02-29`. The evaluation of the expression following `{~}` is carried out as usual.

## 3.2 Longstrings in images

The Longstring datatype is not supported inside images. However, *Sirius Mods* version 7.2 introduces support for image items with length greater than 255.

The following is an example of a declaration of a 300-byte image item:

```
image foo
  bar  is string len 300
end image
```

While such image items can't have arbitrary lengths up to  $2^{31}-1$  like other longstring variables, they exhibit the same behavior as other longstring variables in:

- Request cancellation in the case of truncation
- Upgrading With operations to longstring With operations

## 3.3 Report writer parameters

Report writers can use a lot of virtual storage which can adversely effect other users in an Online. The new CURREP31, CURREP64, HGHREP31, and HGHREP64 system parameters indicate the current and greatest amounts of 31-bit and 64-bit virtual storage used in the *Model 204* run by report writers. The MAXREP31 and MAXREP64 parameters indicate the maximum amount allowed.

The only existing report writer is invoked by the *SirTuneReport* method of the Dataset class. That method is called by the *SirTune* product, and it can also be called directly.

Only the parameters that specify the maximum amount of storage (MAXREP31 and MAXREP64) are resettable.

## 3.4 Janus stream logging

The JANUS TCPLOG subcommand lets you capture all input and output streams for a particular Janus port. The captured streams are written to a sequential file.

One suggested use for a file of captured streams is to provide “playback” for customer-written applications that simulate real workloads during testing of new system or application code.

Details of the format of the log file are available in the *Janus TCP/IP Base Reference Manual*.

### **3.5 SirScan accepts large datasets**

*SirScan* now supports the use of z/OS large datasets for CCAJRNL.

## ***Janus Debugger and Sirius Debugger***

The *Janus Debugger* is a tool designed for software developers who create and maintain &JWEB applications. The *Sirius Debugger* is designed to debug classical screen-based 3270 applications as well as BATCH2 applications. Though the two debuggers use different thread types on the host, they can run concurrently, and they both use the same Windows-based GUI client. For more information about the debuggers, see the ***Janus/Sirius Debugger User's Guide*** on the Sirius web site Documentation page.

The debuggers were first available with Version 6.8 of the *Sirius Mods* and have been steadily enhanced since. Some enhancements are delivered solely by new releases of the GUI client, while other enhancements depend upon features added to new releases of the *Sirius Mods*. Each of the GUI releases is capable of working with all versions of the *Sirius Mods*.

The features described below are introduced with *Sirius Mods* version 7.2 or were added by update builds of the client GUI subsequent to the release of *Sirius Mods* 7.1.

### **4.1 Features available only with Sirius Mods 7.2**

#### **4.1.1 Jump to arbitrary statement**

The Jump feature lets you alter the normal flow of program execution in the Debugger by invoking an immediate transfer of control to a specified statement and then executing that statement. The target statement may be earlier or later in the request than the current statement.

You invoke a jump by right-clicking a line in the Client's Source Code window or by using commands in a macro or mapped button or hot key. The commands offer additional functionality: jumps to a line number, jumps that are a number of lines relative to the current line, and jumps to lines that contain specified strings.

#### **4.1.2 Audit Trail display for procedures excluded by White List processing**

When white listing is on, the Debugger filters procedures automatically, stopping to interactively debug only the requests that are on the white list. A non-listed procedure executes normally, but it is not interactively debugged, and the Debugger Client's Audit Trail displays are immediately refreshed (as of *Sirius Mods* 7.2) to specify that such a procedure has been "skipped."

### 4.1.3 Interrupting White List or Run Until processing

Once the Debugger's White List processing is invoked, it continues in effect unless you explicitly turn it off or until you exit the Debugger Client. It is manually interruptable, however, as of version 7.2 of the *Sirius Mods*: the mappable Client command `breakOnNextProc` lets you override the White List to interactively debug the next procedure.

## 4.2 New features for Sirius Mods 6.9 through 7.2

These features were added by update builds of the Client subsequent to the release of *Sirius Mods* 7.1.

### 4.2.1 Enhanced support for macros

User-created macros were made available for the Debugger in *Sirius Mods* version 7.1. The following subsections describe enhancements to macro support added in *Sirius Mods* 7.2.

#### 4.2.1.1 New macro-only commands

These macro-only commands are added:

**set** Lets you create and initialize macro variables.

**assert** Performs an equality comparison between a) the value of program data or a macro variable and b) a constant or the value of a macro variable or macro function. For example:

```
assert %i=666
```

`assert` failure messages are displayed in the macro console, if it is open. Otherwise, they are displayed in a Windows message box.

**vardump** Displays the values of all current macro variables. The message is normally displayed in a standard Windows informational box (entitled `varDump for Macro`). If the macro console is open, however, the message is sent to the console instead.

**echo** The macro-only command `echo` lets you display a user-specified message. The message is normally displayed in a standard Windows informational box (entitled `Macro message`). If the macro console is open, however, the message is sent to the console instead.

### 4.2.1.2 **Passing a command argument to a macro**

This feature lets you pass an argument to a macro command at the time the macro runs. You can use either a standard macro variable or a standard macro function:

- The `&argstring` macro variable is, in effect, a placeholder for a macro command argument. You specify `&argstring` in a macro where you would normally specify the command argument. Then, depending how you invoke the macro, the Debugger substitutes an explicit value for `&argstring` when the macro runs.

The client obtains the substitute value from the `Variable or Field` text area, from the Macro Command Line tool, or from a mapped `macro` command line.

- The `&&prompt` macro function is used to prompt for a macro command argument. The function format is:

```
&&prompt("promptString")
```

where *promptString* is a double-quoted character string.

When a macro command that contains a `&&prompt` function in place of the command argument is executed, the Client displays a dialog box which uses *promptString* to solicit the value to substitute for the command argument.

### 4.2.1.3 **A console for macro information**

The `Macro Console` is an independent window that logs informational messages about the Debugger macros you run. It is invoked from its option in the Client's `Macros` menu. When a macro is called, the Macro Console displays information about the macro, including its starting and stopping, as well as any error messages.

### 4.2.1.4 **Macro command line**

The `Macro Command Line` menu option opens the Macro Command Line dialog box, which provides a command line interface for running macros. This dialog box stays available unless explicitly closed, so it is a step-saver for macro testing.

### 4.2.1.5 **Macro-running indicator**

A `... macro running` indicator is now displayed on the Client's title bar while a macro is running or waiting.

### 4.2.2 Watching mixed-case global variables

Prior to this feature, you add a global variable to the Client's Watch Window by specifying the variable name (in any case) preceded by "G." or "g". Then the Debugger searched for (only) the variable with the all-uppercase form of the variable's name.

As of *Sirius Mods 7.2*, if you enclose the variable name in single quotation marks, however, the Debugger will search for exactly the case of the characters that you specify.

### 4.2.3 Keyboard-invoked consecutive searches

These two changes to the Search feature enable keyboard-invoked consecutive searches:

- The default response to pressing the Enter key while the Search text box has focus is changed from "search from the top" to "search from current line." This lets you press Enter repeatedly to find subsequent occurrences of a search string. Formerly, pressing Enter again found the same occurrence as the first time, because it repeated the search from the top.

**Note:** This introduces a small upward incompatibility.

- The new `focusToSearchBox` command gives the input focus to the Search string text area. The Ctrl+F key combination is changed to perform the same function by default.

### 4.2.4 New mappable commands

These mappable commands are added:

#### **addWatchOnCurrentLine**

Adds to the Client Watch Window any variables found in the current Source Code line. It is the same as right-clicking a line and selecting Add Watch from the context menu.

The new Data Display menu item `Add Watch on Current Line` has the same effect as this command.

#### **toggleBreakpointOnCurrentLine**

Sets (or removes) a breakpoint for the current Source Code line if the line is or starts an executable statement. It is the same as double-clicking a line or right-clicking a line and selecting `Toggle Breakpoint` from the context menu.

The new Breakpoints menu item `Toggle Breakpoint on Current Line` has the same effect as this command.

**reloadWhiteList**

Updates the existing White List with the current contents of the `whitelist.txt` file, so you can dynamically update your White List. Same as clicking the `Reload White List` button on the Proc Selection page.

The new Execution menu item `Reload White List` has the same effect as this command.

**turnOnWhiteList, turnOffWhiteList**

Activates or deactivates White List filtering (like clicking the `Turn On White List` button or the `Turn Off White List` button on the Proc Selection page).

The new Execution menu items `Turn On White List` and `Turn Off White List` have the same effect as these commands.

**runUntil** Runs program code without interruption until it reaches a specific procedure, then displays that procedure for debugging. Its required argument is the name of, or a character pattern for, the target procedure.

Having the same effect as the `Run Until Procedure` button., this command was formerly a macro-only command.

The new Execution menu item `Run Until Proc` has the same effect as this command.

## 4.2.5 ON UNIT code debugging

An exception to ordinary request-cancellation handling is added for the purpose of ON UNIT code debugging. Instead of stopping you from stepping through a request when the request has a canceling error, the Client lets you continue in the exceptional case of the following error only (which occurs within a User Language ON UNIT):

```
M204.1982: ILLEGAL JUMP ATTEMPTED OUT OF COMPLEX SUBROUTINE ON UNIT
```

## 4.2.6 Reporting for DEBUGGERSERVER and DEBUGGERCLIENT Janus ports

The DEBUGGERSERVER and DEBUGGERCLIENT Janus ports defined for the Debuggers are now reported as DEBUGSRV and DEBUGCLI ports, respectively, in the output from JANUS STATUS and JANUS DISPLAY commands. Formerly, they were reported as SRVSOCK and CLSOCK types.



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

## 5.1 The Dataset class

The Dataset classes provide an object-oriented interface to sequential datasets. This interface is more flexible than the traditional image-oriented interface used in User Language. In addition, they provide User Language access to *Model 204* streams — datastreams composed of one or more sequential datasets via the *Model 204* DEFINE STREAM command (see the *Model 204 Command Reference Manual*).

The Dataset class operates on datasets or streams defined by any of the following:

- An MVS DD card.
- A VSE DLBL card.
- A CMS FILEDEF command.
- A *Model 204* ALLOCATE command (see the *Model 204 Command Reference Manual*).
- A *Model 204* DEFINE STREAM command (see the *Model 204 Command Reference Manual*).

The Dataset class uses two class-specific enumerations: the RecordFormat enumeration, and the DatasetState enumeration.

### 5.1.1 Dataset example

The following simple example is a program that displays the contents of sequential dataset (or stream) SEQIN:

```
b
%ds  is object dataset
%l  is longstring

%ds = new('SEQIN')
%ds:open

repeat forever
  %l = %ds:readRecord
  if %ds:state eq afterEnd then
    loop end
  end if
  print %l
end repeat

end
```

### 5.1.2 RecordFormat Enumeration

The RecordFormat enumeration indicates the format of the blocks in the sequential dataset. While, strictly speaking, the RecordFormat enumeration actually describes block formats or maybe file formats, it corresponds to the RECFM value used in DD cards, ALLOCATE statements, etc., so the term RecordFormat was used. The values of the RecordFormat enumeration are:

- U** Undefined format (each block is treated as a single record).
- F** Fixed format (each block contains one fixed length record).
- V** Variable format (each block contains one variable length record).
- FB** Fixed blocked format (each block contains one or more fixed length records).
- VB** Variable blocked format (each block contains one or more variable length records).

### 5.1.3 The DatasetState Enumeration

The DatasetState enumeration describes the current state of a Dataset object. A newly created Dataset object starts in the Closed state and then changes state as certain methods are performed on it. Certain methods are restricted to Dataset objects in a particular subset of states. The values of the DatasetState enumeration are:

**Closed** Object has just been created or just closed.

**Open** Object has been opened and can either be read from or written to.

**AfterEnd** Last read from object detected end of file.

**Full** Last write to object detected a file full condition.

### 5.1.4 Dataset methods

The following dataset methods are available:

**Blocksize property** Indicates the blocksize for the underlying dataset.

**Close subroutine** Closes an open dataset.

**Discard subroutine** Discards a dataset object.

**New constructor** Creates a new dataset object, associating it with a sequential dataset or stream.

**NumberOfBuffers property** Indicates the number of virtual storage buffers to be used, or being used, to read or write blocks of data.

**Output property** Indicates whether or not the dataset is to be used for, or is being used for, output (written to).

**ReadBlock function** Reads a block of data from the dataset.

**ReadRecord function** Reads a record from the dataset.

**RecordFormat property** Indicates the record and block format for the underlying dataset.

**RecordLength property** Indicates the record length for the underlying dataset.

**SirtuneReport function** Reads a *SirTune* sample dataset and produces a report (in XML format) summarizing the data in that sample dataset. This function is only available to sites authorized for *SirTune*.

**WriteBlock subroutine**      Writes a block of data to the dataset.

**WriteRecord subroutine**    Writes a record to the dataset.

## 5.2    **Intrinsic classes**

User Language is a legacy programming language for which object-oriented capabilities are a relatively recent addition. So, one might expect that strings and numbers are not maintained internally as objects, since their existence pre-dates objects. This is in fact the case, though as has been shown, this is largely irrelevant from a User Language programmer's perspective. And even if object-oriented capabilities were present in User Language from its inception, strings and numbers might have been special-cased for efficiency anyway.

However, beyond the internal representation of strings and numbers, a distinction between User Language and pure object-oriented languages was that the object:method syntax was not used to manipulate strings and numbers; instead, strings and numbers were manipulated via \$functions. In *Sirius Mods 7.2*, this has been changed to allow the object:method syntax against string and numeric variables and constants.

The following code fragment illustrates how the same operation can be accomplished via traditional \$functions and via object-oriented syntax:

```
%name    is string len 32
...
print $len(%name)
print %name:length
```

While this might not seem very significant, it provides considerable value:

- It allows User Language to be used as a “pure” object-oriented language. This might be especially appealing to programmers who were trained in a pure object-oriented language.
- It provides the benefit that expressions can generally be read in the natural left-to-right manner rather than the inside-to-outside manner required to understand an expression coded with \$functions.
- It provides capabilities to methods that operate on strings or numbers that are not available with \$functions. These include support for named parameters and the ability to take objects as input parameters and to produce objects as results.

While it would have been possible to extend \$functions to have this same functionality (much as *Sirius Mods* provided callable \$function support), it makes more sense to provide it using true object-oriented syntax. Given that this has now been done, it is unlikely that these capabilities will ever be added to \$functions.

User Language traditionally allowed declarations of many different datatypes. These included Strings of a specified length with a possible DP value, Fixed numerics (also with a possible DP), and Float numerics. In addition, *Sirius Mods* provided support for Longstring datatypes. And images provide support for a variety of additional datatypes, including packed and zoned formats. Essentially, however, there are two categories of datatypes in User Language: string and numeric. In User Language one can easily assign values from one datatype to another, even between numeric and string types. This is intuitive and useful. Typically, in the “real world” one doesn't distinguish between the string representation of a number and its numeric value used for calculation. Similarly, except for performance purposes and, perhaps, value limits, one is typically not concerned about how a value is stored internally.

This is not the case in some “pure” object-oriented languages where the paradigm of strong-datatyping is extended to numeric and string datatypes. This means that if one wants to display the value of a numeric variable, one must explicitly convert it to a string (since what one displays is a string). Similarly, if one wishes to do a calculation using a value in a string (perhaps read from an external data source), one must explicitly convert the string to a number. It is asserted here that strong-datatyping for strings and numbers is a mistake, allowing some vision of purity to prevent programmers from doing something completely natural (treating numbers as strings, and strings as numbers) and something that is done hundreds of times in any program of any size. This view is reinforced by the fact that the general industry trend is away from strong-datatyping for strings and numbers and toward implicit conversion between these types.

It is worth pointing out, however, that loose-datatyping between strings and numbers does not imply loose-datatyping between strings and numbers and other classes. That is, there are generally no implicit conversions of strings or numbers to or from non-string, non-numeric objects.

Because loose-datatyping and the existing User Language datatypes work quite well in facilitating rapid development of reasonably tight and efficient code, the *Janus SOAP ULI* support for methods against string and numeric types does not add any new datatypes to User Language. And, because of loose-datatyping, the numeric and string methods can be applied to any of the standard User Language string and numeric datatypes. Because they apply to these intrinsic User Language datatypes, these methods are called **intrinsic** methods.

Intrinsic methods can be further classified into three subsets:

- Float** Methods that perform numeric manipulation on values in *Model 204's* Float format.
- Fixed** Methods that perform numeric manipulation on values in *Model 204's* Fixed format (with an assumed DP of 0).
- String** Methods that perform string manipulation, with Longstring capability assumed.

The Float intrinsic class can really be thought of as a generic numeric class, as it is a convenient way of representing both integer and decimal data. This is different from most other programming languages where a Float datatype suffers from inconvenient behavior, especially for decimal numbers. The Float class is so convenient that all numeric parameters to system methods and \$functions are treated as Float parameters. Also, because of the convenience of the Float class, there are currently no methods in the Fixed class.

As such, the intrinsic methods can be thought of as belonging to two intrinsic classes: the *Numeric* class and the *String* class. These classes behave largely as if their inputs were User Language Float and Longstring datatypes, respectively. That is, for all intents and purposes, the following are true:

- Any non-Float values are converted to Float values before being processed by a Numeric method. This includes the conversion of any non-numeric strings into 0 that occurs in other User Language contexts that take a numeric input.
- For the purposes of truncation and the With operation, all String method string inputs or outputs behave as longstrings.

### 5.2.1 Intrinsic method objects

The term “method object” (or “intrinsic method object”) is used for the value or variable to which an intrinsic method is applied, even though the value or variable isn't really an object. This is justified because, as noted above, strings and numbers can be considered immutable objects, regardless of their history or internal representation.

For example, in the following case, `%str` is the intrinsic method object for the `Length` method:

```
%str is string len 32
...
print %str:Length
```

Intrinsic methods can be applied to constants, in addition to variables. For example, the following assigns the length of the string literal “Whatever” to `%x`:

```
%x = 'Whatever':length
```

Intrinsic methods can, of course, take the output from another method, intrinsic or not, as its input: For example, the following uses the `Right` method (which gets the rightmost characters of a string) against the output of a `Stringlist Item` method:

```
%list is object stringlist
...
%value = %list:item(%i):right(8)
```

Of course, since it is unnecessary to specify the Item method for a Stringlist, the above can also be written as:

```
%list    is object stringlist
...
%value = %list(%i):right(8)
```

Intrinsic methods can also be applied to image and screen items:

```
image michigan
  romulus  is string len 32
...
end image
...
%value = %michigan:romulus:right(8)
```

Intrinsic methods can also be applied to the output from a \$function:

```
%seconds = $time:right(2)
```

Intrinsic methods can even be applied to the results of expressions:

```
...
%value = (%tweedledum with %tweedledee):right(8)
```

As with other methods, the colon that separates the method object from the method name can be optionally preceded or followed by spaces. This could be done to enhance readability, or even to split a long line. The following four statements are all equivalent:

```
%value = %list:item(%i):right(8)
%value = %list: item(%i): right(8)
%value = %list :item(%i) :right(8)
%value = %list : item(%i) : -
                right(8)
```

As with most other uses of intrinsic variables or values, if the method object is of a different type than the data type on which the method operates, the input value is automatically converted into the target datatype. For example, the expression  $7/3$  clearly produces a numeric value, but the Left method operates on strings. So, in the statement:

```
%i = (7/3):left(4)
```

the result of the division is converted into a string and then passed to Left, which produces `2.33` as its result.

Because of this automatic conversion, the specific class (String or Float) of an intrinsic method cannot be determined from the datatype of its method object. This means that the class must be determined only from the name of the method, which means that

method names must be unique among all intrinsic classes (that is, among String, Float, and Fixed). For example, there may not be a Length method in both the String and Float intrinsic classes. In the case of Length, the method is an intrinsic String method.

## 5.2.2 Intrinsic method syntactic oddities

User Language is a legacy programming language that supports some unusual syntax. Some of this syntax causes problems for the use of intrinsic methods, as is described for the following three cases:

- Intrinsic methods against database field names

Field names have very loose naming rules, and names can contain colons and spaces. Because of this, a field name must be contained inside of parentheses to be used as a method object. For example, the field `big fat greek field` is used as the input to the intrinsic Substring method:

```
for each record in %recordset
  ...
  %value = (big fat greek field):substring(2, 3)
  ...
end for
```

- Intrinsic methods against percent variables and images that have the same name

User Language allows one to declare images and percent variables with the same name in the same scope. That is, inside a method, one can declare an image called `holland` and a %variable called `%holland`. References to items in the image and to the percent variable both start with `%holland`:

```
image holland
  ...
  factory is float
  ...
end image
...
%holland is string len 10
...
%a = %holland:factory
...
%a = %holland
```

Historically, this has not been a problem because a percent variable would never be followed by a colon. However, with intrinsic methods, this is no longer the case. If the Holland image contained an item called Length, it would be impossible to tell whether `%holland:length` referred to the Length item in the Holland image or the intrinsic Length method applied to `%holland`.

Because of this ambiguity, intrinsic methods cannot be used against percent variables without a blank between the variable name and the colon if there is an image with the same name as the percent variable. This is true regardless of whether or not there is an actual conflict between the method name and an image item name.

So, using the above example, to apply the length method to `%holland`, you must do one of the following:

```
%a = %holland :length
%a = %holland : length
```

Specifying `%holland: length` would not work. A space after the variable name indicates that the reference is *not* to the image because image item references do not allow any blanks between the image name, colon, and image item name.

You can also simply wrap the variable name in parentheses:

```
%a = (%holland):length
```

Of course, the best solution is to avoid using the same name for images and percent variables, as this is generally somewhat confusing, anyway.

- Intrinsic methods in a Print, Audit, or Trace statement

The Print, Audit, and Trace statements all use somewhat unusual syntax. While initially these statements appear to operate on expressions, just like an assignment, this is not really the case. For example, the following statement gets a compilation error:

```
print 3*4
```

Because of this, one has to be careful using intrinsic methods in a Print, Audit, or Trace statement. First, because blanks are treated specially in these statements, one can't put a blank before the colon in an intrinsic method invocation. That is, the following is **incorrect**:

```
print %x :length
```

But the following two statements are allowed:

```
print %x:length
print %x: length
```

In addition, string and literal constants are treated specially by these statements, so one cannot issue methods against constants in a Print, Audit, or Trace statement. That is, the following are **incorrect**:

```
print 'Foobar':length
print 22:squareRoot
```

Finally, one might be tempted to use parentheses to get around some of these issues, but, unfortunately, leading parentheses are not allowed in a Print, Audit, or Trace statement token. That is, the following is **incorrect**:

```
print (fieldname):length
```

Coupled with the fact that applying intrinsic methods to fields generally requires use of parentheses, this means that one **cannot** display the result of an intrinsic method applied to a field with the Print, Audit, and Trace methods.

Fortunately, *Sirius Mods* version 7.2, the same version that introduced intrinsic methods, also introduced the PrintText, AuditText, and TraceText statements. These are direct analogs of Print, Audit, and Trace, respectively, but they use a more consistent syntax. Specifically, the newer statements treat everything as literal text, except for that which is enclosed by curly braces (`{ }`), which is treated as a standard User Language expression. This means that the syntax used for variable parts of PrintText, AuditText, and Tracetext statements is identical to the syntax allowed on the right side of a variable assignment.

So, the following is valid:

```
printText {3*4}
```

And this is valid:

```
printText {%x :length}
```

These are valid statements:

```
printText {'Foobar':length}  
printText {22:squareRoot}
```

And this is valid:

```
printText {(fieldname):length}
```

The Text statement also provides functionality comparable to the PrintText, AuditText, and TraceText statements, and it is especially useful for displaying multiple lines of data.

So it is recommended that you discontinue the use of the Print, Audit, and Trace statements in favor of PrintText, AuditText, and TraceText, as of *Sirius Mods* 7.2.

### **5.2.3 Intrinsic Float methods**

The individual intrinsic Float methods are described in the following sections.

Float intrinsic methods treat their method object as a Float value. Any value that is not a Float value is automatically converted before it is acted on by the method.

### 5.2.3.1 **SquareRoot function**

This function returns a number that is the square root of the method object. An attempt to get the square root of a negative number results in request cancellation.

```
%value = number:squareRoot
```

#### **SquareRoot syntax**

### 5.2.4 **Intrinsic String methods**

The individual intrinsic String methods are described in the following sections.

Intrinsic String methods treat their method object as a Longstring value. Any value that is not a String value is automatically converted before it is acted on by the method.

#### 5.2.4.1 **Base64toString function**

Base-64 encoding is useful for encoding strings that might contain binary or other characters that could cause problems in certain contexts. This function returns the unencoded value of a base-64 encoded string.

```
%out = string:base64toString
```

#### **Base64toString syntax**

#### 5.2.4.2 **HexToString function**

This function returns the unencoded value of a hex encoded string. Hex (short for hexadecimal) encoding is usually used for debugging when there is a concern that non-displayable characters (including trailing blanks) might be present in a string. By hex encoding such a string, all non-displayable bytes are converted to displayable hexadecimal equivalents.

```
%out = string:hexToString
```

#### **HexToString syntax**

### 5.2.4.3 **Left function**

This function returns the left-most characters of the method object string, possibly padding it on the right.

```
%out = string:left(length, [Pad=char])
```

#### **Left syntax**

### 5.2.4.4 **Length function**

This function returns the length in bytes of the method object string.

```
%len = string:length
```

#### **Length syntax**

### 5.2.4.5 **PositionIn and PositionOf functions**

These functions return the numeric position of the first occurrence of one string inside another. The difference between the two methods is that for PositionIn, the method object string is located in the first argument string, whereas for PositionOf, the first argument string is located in the method object string. Which is more convenient to use will be application dependent.

```
%pos = needle:positionIn(haystack, [start=spos])  
%pos = haystack:positionOf(needle, [start=spos])
```

#### **PositionIn and PositionOf syntax**

### 5.2.4.6 **RegexMatch function**

This function determines whether a given pattern (regular expression, or “regex”) matches within a given string according to the “rules” of regular expression matching (information about the rules observed is provided in the ***Janus SOAP Reference Manual***).

```
%pos = string:regexMatch(regex [,Options=opts])
```

#### **RegexMatch syntax**

### 5.2.4.7 **RegexReplace function**

This function searches a given string for matches of a regular expression, and it replaces matches with, or according to, a specified replacement string. The function stops after the first match and replace, or it can continue searching and replacing until no more matches are found.

Matches are obtained according to the “rules” of regular expression matching (information about the rules observed is provided in the *Janus SOAP Reference Manual*).

```
outStr = string:regexReplace(regex, replacement -  
                             [,Options=opts])
```

#### **RegexReplace syntax**

### 5.2.4.8 **RegexSplit function**

This method repeatedly applies a regular expression, or “regex,” to the method object string until it has tested the entire string. This splits the string into the substrings that are matched by the regex (the “separators”) and the substrings that are not matched. By default, the method saves each *unmatched* substring as a separate item in the StringList result object. The leftmost unmatched substring is the first item in the StringList, the next leftmost is the second item, and so on.

A simple application of the method is to extract only the data items from a string of comma-separated data items. If the specified regex is a comma, each of the resulting StringList items will contain one of the data items.

RegexSplit uses the rules of regular expression matching (information about which is provided in the *Janus SOAP Reference Manual*).

```
outList = string:regexSplit(regex          -  
                            [,Options=opts] -  
                            [,Add=output])
```

#### **RegexSplit syntax**

#### 5.2.4.9 **Right function**

This function returns the right-most characters of the method object string, possibly padding it on the right.

```
%out = string:right(length, [pad=char])
```

##### **Right syntax**

#### 5.2.4.10 **StringToBase64 function**

This function returns the base-64 encoded value of the method object string.

```
%out = string:StringToBase64
```

##### **StringToBase64 syntax**

#### 5.2.4.11 **StringToHex function**

This function returns the hex encoded value of the method object string.

```
%out = string:StringToHex
```

##### **StringToHex syntax**

#### 5.2.4.12 **Substring function**

This function returns a specific number of bytes starting at a specific position in the method object string, possibly padding it on the right.

```
%out = string:substring(start, length, [Pad=char])
```

##### **Substring syntax**

### 5.3 **Exception handling**

To diagnose or correct or reduce the damage from run-time errors that occur in places in a program where the use of a request cancellation, return-value processing, or output-parameter processing are insufficient, the *Janus SOAP ULI* introduces exception handling.

Exception handling consists of three parts:

1. The ability to define classes of exceptions and the information available for these classes. Unsurprisingly, classes of exceptions are defined in almost the same way as other classes.
2. The ability to indicate an unusual or exception situation. This is referred to as *throwing* an exception.
3. The ability to detect a thrown exception and to perform special processing for the exception. This is known as *catching* an exception.

Although the *Janus SOAP ULI* exception handling support is very similar to many other object-oriented programming languages, such as Java or VB.Net, each language's implementation of exception handling has subtle or not so subtle differences from the others.

As a significant example, while many object-oriented languages make all (or almost all) errors exceptions, *Janus SOAP ULI* does not. This is partially because many object-oriented languages are not, strictly speaking, application languages — in addition to writing applications in some other object-oriented languages, one might write programming environments or even operating systems. As such, a programming environment or operating system must be able to intercept all errors and try to deal with them, so the object-oriented languages facilitate this by making all errors catchable.

User Language is an application development language. While it is theoretically possible to build a program environment in User Language, or even an operating system, this is not really the focus of User Language, and it is not likely anyone would ever do this. The User Language programming environment (*Model 204*, which is not, itself, written in User Language) is responsible both for the ultimate catching of errors and the providing of information about the nature of the errors so that they can be corrected.

The kinds of errors that *Janus SOAP ULI* does *not* turn into exceptions, and thus are not trappable, fall into two broad categories:

1. Pernicious environmental errors from which it would generally be almost impossible to recover gracefully on an application level.

For example, if CCATEMP fills up, it is exceedingly unlikely that a request could recover gracefully. Since CCATEMP is used for so many different things (including record sets for finds, stringlists, collections, internal storage of objects swapped out of a server, transaction backout logs, and so on), it would be very difficult for an application to anticipate all the places that CCATEMP could fill. And even if it could anticipate the places, it would be very difficult to avoid doing anything that might also require CCATEMP.

2. Errors that are indicative of logic errors in a program.

For example, a class member reference via a null object variable is usually indicative of a programming error. Similarly, a reference to an invalid collection item number is also generally indicative of a programming error.

While there might be error-causing cases of “sloppy” references to object variables or collection items that you would want to simply catch when they happen:

- Most of these are handled well by existing facilities such as Auto New and also UseDefault processing for collections.
- In the odd cases where such sloppy references are useful, it is trivial and efficient to check for the sloppy references (null object variable, invalid collection item number), so an exception-handling paradigm adds little value.

### 5.3.1 Exception class definitions

The collection of data describing an error situation might be used immediately after the error is detected, or it might be examined elsewhere. As such, this error data should be able to persist indefinitely. Given this requirement, it is clear that the logical place to hold error information is inside an object. And the description of the data associated with a particular error would be a class.

Classes that describe trappable error situations are called **exception classes**. To a large degree, exception classes are no different from any other class:

- They can have Public, Private, and Shared blocks; they can have variables, methods, and the same Allow and Disallow rules as any other class.
- There can be system exception classes, which are maintained by the *Janus SOAP ULI* environment, and there can be User Language exception classes, which are defined and maintained by User Language code.
- Exception classes can be used wherever non-exception classes are used, although the reverse is not true. That is, there are certain statements that require an exception class or an object of an exception class. This is largely to prevent accidental misuse of a non-exception class in an exception class context.

User Language exception classes are denoted by using the `Exception` keyword in the class header:

```
class tackyColor exception
```

One can think of the `Exception` keyword on the Class declaration as indicating that the class extends some hidden base class.

You can put the `Exception` keyword on almost all class declarations; specifying it removes no capabilities from the class. Doing so, however, is misleading, since most

classes are likely never to be used to indicate error situations. In this regard, you can view the `Exception` keyword on a class declaration as documentation that a class can be used as an exception class. While the compiler *cannot* ensure that a class with an `Exception` declaration will be used as an exception, it *does* ensure that a class not declared as `Exception` is not used in exception contexts.

The one case where the `Exception` keyword is not allowed on class declarations is in declarations that extend non-exception classes — exception classes can only extend other exception classes.

Some examples of system exception classes (described further in “[System Exception classes](#)” on page 38) are:

- InvalidHexData** Describes an error in converting hexadecimal data to some other datatype.
- InvalidRegex** Describes an error processing a regular expression.
- DaemonLost** Describes a situation where the thread doing processing for a Daemon object was lost (logged off).

### 5.3.2 Throwing exceptions

When an error situation occurs, the code that detects the situation can do one of two things:

- It can cancel the request. In User Language this is done with an `Assert` statement. In system code, this is done by some equivalent of the `Assert` statement. This is the correct response to an error if the error is clearly indicative of a programming error or a severe environmental constraint that is likely to make request continuation unproductive.
- It can *throw* an exception. In User Language this is done with a `Throw` statement. In system code, this is done by some equivalent of the `Throw` statement. This is the correct response to an error if the error might occur in the normal course of processing and the code that called the method might be able to recover from the error.

**Note:** In the *Janus SOAP ULI* implementation of exceptions, exceptions can only be thrown by methods.

In both system and User Language methods, a thrown exception can only be handled (caught) by the code that called the method, as opposed to inside the same method that threw the exception.

The exceptions that might be thrown by a method must be documented in the method header.

### 5.3.2.1 Specifying a Throws clause

The exceptions thrown by a method are indicated by the `Throws` clause in the method declaration and definition header. For example, if the method `Paint` might throw a `TackyColor` and `InvalidHexData` exception, it should be declared and defined as:

```
subroutine paint(%hexColor is string len 6) -  
    throws tackyColor and invalidHexData
```

The keyword `and` is used to separate the exceptions that might be thrown by a method. The method can only ever throw one exception at a time, and in most cases, it will not throw an exception at all.

The list of exceptions after a `Throws` keyword is the list of exception class names. These class names could be User Language exception class names, or they could be system exception class names. If the class is a system class name, it could be fully qualified with the `System:` namespace indicator. For instance, the previous example could be written as

```
subroutine paint(%hexColor is string len 6) -  
    throws tackyColor and -  
    system:invalidHexData
```

As with most method descriptions, the `Throws` clause on a method declaration and definition must match exactly.

Methods that implement an overridable method cannot throw any exceptions not thrown by the implemented method; however, they do not necessarily have to throw all the exceptions thrown by the implemented method. For example, if an overridable method in class `Products` indicates:

```
subroutine buy(%productCode is string len 8) -  
    overridable -  
    throws tooExpensive and outOfStock
```

then it is valid for an implementing method to indicate

```
subroutine buy(%productCode is string len 8) -  
    implements buy in products -  
    throws tooExpensive
```

However, it is **not** valid for the implementing method to throw an exception other than `TooExpensive` or `OutOfStock`.

### 5.3.2.2 Using the Throw statement

Once a method declaration indicates the exceptions it might throw, that method could then throw the exception with the `Throw` statement. The `Throw` statement must be

followed by an instance of the exception class being thrown. For example, if the method header contains

```
subroutine paint(%color is string len 6) -  
    throws tackyColor
```

and there is a variable declaration in the method

```
%tacky    is object tackyColor
```

the method could do

```
throw %tacky
```

Of course, the Throw statement is likely to be inside an If clause, since exceptions are generally thrown in unusual situations, not common ones:

```
if %color eq %yuckyGreen or    -  
    %color eq %grottyOrange then  
    throw %tacky  
    auditText Threw an exception  
end if
```

Note that in the above example, the auditText statement after the Throw will **never** be executed. This is because the Throw either returns immediately to the method caller (if the method caller catches the exception), or it cancels the request immediately.

Another problem with the above example is that %tacky was likely to never have been set to reference a TackyColor instance. So, more correct would be something like:

```
if %color eq %yuckyGreen or    -  
    %color eq %grottyOrange then  
    %tacky = new  
    throw %tacky  
end if
```

But, even this is not quite right. Usually, the exception objects will contain information to aid in problem determination and recovery:

```
if %color eq %yuckyGreen then  
    %tacky = new  
    %tacky:reason = 'Yucky'  
    %tacky:alternative = '00FF00'  
    throw %tacky  
end if  
if %color eq %grottyOrange then  
    %tacky = new  
    %tacky:reason = 'Grotty'  
    %tacky:alternative = 'FFA500'  
    throw %tacky  
end if
```

This example illustrates the point that it is common for any `Throw` of a particular exception class to always return more or less the same information. As such, exception classes often have constructors that can specify all the information provided by the class. Then, no variables of the exception class need to be defined; the result of the constructor call can simply be thrown:

```
if %color eq %yuckyGreen then
    throw %(tackyColor):new(reason='Yucky', -
                            alternative='00FF00')
end if
if %color eq %grottyOrange then
    throw %(tackyColor):new(reason='Grotty', -
                            alternative='FFA500')
end if
```

An attempt to throw an exception object whose class does not match one of the classes listed in the method declaration's `Throws` clause results in a compilation error.

### 5.3.2.3 Exception classes extending other exception classes

Exception classes can extend other exception classes. As such, the class of an object specified in a `Throw` statement does not have to match any class in the method's `Throws` list exactly — it can be of an extension class of one of the `Throws` list classes. Because an extension class can, itself, be extended, and because of multiple inheritance, this means that a `Thrown` exception object might match multiple classes in a method's `Throws` list.

The thrown (and so catchable) class is the first class in the `Throws` list that matches the object in the `Throw`. That is, the first class in the `Throws` list that exactly matches the thrown object's class or that is a base class of the thrown object is used as the thrown class for the method caller.

### 5.3.3 Try and Catch

Without any action on a method caller's part, a thrown exception is, for all intents and purposes, a request cancelling error. To prevent the request cancellation, an exception must be “caught.” This is achieved by the use of a **Try/Catch block**.

A `Try/Catch` block consists of two parts. The first, the `Try` section contains one or more User Language statements that might result in an exception. The `Try` section is then followed by one or more `Catch` sections, each one of them handling (catching) a particular class of exception.

The following fragment illustrates the use of a Try/Catch block to trap an exception caused by invalid hexadecimal data:

```
try
  %binary = %input:hexToString
catch invalidHexData
  %binary = '???'
catch invalidBase64Data
  %binary = '???'
end try
```

This example illustrates a few points:

- The end of the statements whose exceptions are being caught is indicated by the first Catch block.
- The class of exceptions being caught follows the Catch statement.
- A catch block is terminated by another Catch statement or an End Try.
- One can catch multiple exception types, each within its own Catch block.
- There is no validation that the type of exception being caught might actually be thrown inside the Try block.

Although a Try block can contain more than one statement, as a general rule of thumb, you should place as few statements inside a try block as possible. To facilitate this, you can follow the Try statement by a User Language statement on the same line, as in:

```
try %binary = %input:hexToString
catch invalidHexData
  %binary = '???'
end try
```

As noted in [“Using the Throw statement” on page 32](#), what gets thrown with an exception is an exception object that contains information about the nature of the exception. To reference the thrown exception object, you must specify a To clause, followed by an object variable of the exception class being caught.

For example, if %daemon is a Daemon object and %daemonLost is an object of the DaemonLost exception class, the following block catches the exception thrown if the daemon thread was logged off for some reason, and it displays the output of the last command up to the point where it logged off:

```
try %daemon:run('INCLUDE NASTY')
catch daemonLost to %daemonLost
  printText Daemon died! Its last words were:
  %daemonLost:daemonOutput:print
end try
```

### 5.3.3.1 Some differences with other languages

The following are some differences between the *Janus SOAP ULI* implementation of Try/Catch and implementations in other languages. Outside of these differences, Try/Catch support in *Janus SOAP ULI* is very similar to that in other languages.

- Unlike Java, it is not necessary to provide a Catch for all exceptions that a method might throw.
- Unlike many other languages, uncaught exceptions are not automatically propagated to higher level callers. Partially, this is because the User Language environment is not written in User Language, so there is no need to propagate exceptions to some outer User Language environment which, presumably, would clean up the failing request and possibly provide diagnostics about the error. Instead, clean-up and diagnostics are provided by the assembler environment.
- The absence of automatic exception propagation eliminates the utility of a `Finally clause` (to “ensure” that the method doesn't leave things in a half-done state), so no Finally clause is available in Try/Catch blocks in User Language.
- Catches cannot catch locally thrown exceptions. That is, a Throw statement always results in immediate exit from the current method, regardless of whether or not the Throw is inside of a Try block and whether or not there are Catches that correspond to the thrown exception.

### 5.3.3.2 Nesting Try/Catch blocks

Like in other languages, Try/Catch blocks can be nested. That is, a Try/Catch block can be inside the try or catch clause of another Try/Catch block:

```
try
  %dmn:run('I STEP1')
  try
    %str = %hex:hexToString
    %dmn:run('I STEP2 ' with %str)
  catch invalidHexdata
    %dmn:run('I STEP2 ???')
  end try
catch daemonLost
  auditText My daemon's gone!
  try %str = %hex:hexToString
  catch invalidHexdata
    %str = '???'
  end try
  return %str
end try
```

The catch that applies to a particular thrown exception is the first Catch that either exactly matches the class of the thrown exception, or is a base class of the thrown exception.

### 5.3.4 OnThrow and OnUncaught

An exception class might want to perform special processing at the time an exception is thrown:

- It might want to make sure the exception object has valid data.
- It might want to record diagnostic information, perhaps to the audit trail or perhaps to some *Model 204* file.
- It might want to derive some variable values that might not necessarily have been derivable in the constructor.

To provide this capability, *Janus SOAP ULI* special-cases two method names in a User Language exception class: `OnThrow` and `OnUncaught`. Both of these methods must be subroutines (as opposed to Functions or Properties) and cannot have parameters.

The `OnUncaught` subroutine is automatically called whenever an exception of the containing class is thrown, and the exception will not be caught. The `OnThrow` subroutine is automatically called whenever an exception of the containing class is thrown, and either the exception will not be caught and there is no `OnUncaught` method in the class, or the exception will be caught.

These two method names have no meaning in non-exception classes.

These methods can be called explicitly, and they can be either Private or Public (though whether they are Public or Private is irrelevant for implicit calls when an exception is thrown).

The following illustrates an `OnThrow` subroutine that makes sure the exception data is valid at the time an exception is thrown:

```
class pratfall exception
  public
    variable sound is string len 32
    subroutine onThrow
  end public
  subroutine onThrow
    assert %this:sound eq 'splat' or -
          %this:sound eq 'boing'
  end subroutine
end class
```

The following illustrates an `OnUncaught` subroutine that logs information from the exception to the audit trail before allowing the request to be cancelled:

```
class pratfall exception
  public
    variable sound is string len 32
    subroutine onUncaught
  end public
  subroutine OnUncaught
    auditText Taking a pratfall -- {%this:sound}
  end subroutine
end class
```

If *SirFact* is available and capturing dumps for requesting cancelling errors, all the information one would need is likely to be in the dump, so there is probably little need to collect extra data in an `OnUncaught` subroutine.

There is no way for an `OnUncaught` or `OnThrow` subroutine to undo the effect of the exception, that is, to prevent a request cancellation if the exception is uncaught. Both routines, however, **can** force a request cancellation, perhaps by using an `Assert` statement, even if the exception would have been caught. If a request cancellation occurs inside an `OnThrow` subroutine for an exception that's about to be caught, the catching statements are not executed, because the request is cancelled before the `OnThrow` subroutine returns.

The `Throws` clause is invalid on an exception class's `OnThrow` and `OnUncaught` subroutines, so these subroutines cannot, themselves, throw an exception.

## 5.4 System Exception classes

As described in “[Exception handling](#)” on page 28, classes that describe trappable error situations are called exception classes. System exception classes are maintained by the *Janus SOAP ULI* environment, while User Language exception classes are defined and maintained by User Language code.

The system exception classes added in *Sirius Mods 7.2* are described in the following sections.

### 5.4.1 DaemonLost class

The DaemonLost exception class indicates that a daemon thread associated with a daemon object was lost, most probably because of a user restart.

The methods of the DaemonLost exception class are:

**DaemonOutput** This readOnly property returns a stringlist that contains a copy of the output of the last stream the daemon thread was running before it was lost (probably restarted). The contents of this stringlist might be useful in determining why the daemon thread was restarted. This stringlist can be empty, that is, have no items.

**New** This callable constructor generates an instance of a DaemonLost exception. As shown below, the required argument of the New method is a setting of the DaemonOutput property.

To produce a DaemonLost exception, you typically use a User Language Throw statement with a DaemonLost New constructor. The New method format follows:

```
[% (DaemonLost): ]New(DaemonOutput=outputList)
```

#### New constructor syntax

*outputList* above is a stringlist that is to be copied to the DaemonOutput property stringlist.

#### Example

The following statements throw a DaemonLost exception, setting the DaemonOutput stringlist to a single item that contains the text *Nearer My God to Thee*:

```
%errorList = new
%errorList:add('Nearer My God to Thee')
throw %(daemonLost):new(daemonOutput=%errorList)
```

### 5.4.2 InvalidBase64Data class

The InvalidBase64Data exception class describes an exception associated with finding non-base64-encoded data where base64-encoded data was expected, usually when decoding base64-encoded data.

The methods of the InvalidBase64Data exception class are:

**Position** This readOnly property returns the position in the (expected) base64-encoded string where a non-base64-encoding character was found.

**New** This callable constructor generates an instance of an `InvalidBase64Data` exception. As shown below, the required argument of the `New` method is a setting of the `Position` property.

To produce an `InvalidBase64Data` exception, you typically use a User Language Throw statement with an `InvalidBase64Data` `New` constructor. The `New` method format follows:

```
[%(InvalidBase64Data):]New(Position=pos)
```

#### **InvalidBase64Data constructor syntax**

*pos* above is the numeric value to be assigned to the exception object's `Position` property.

#### **Example**

The following statement throws an `InvalidBase64Data` exception with the position set to 2:

```
throw %(invalidBase64Data):new(position=2)
```

### **5.4.3 InvalidHexData class**

The `InvalidHexData` exception class describes an exception associated with finding non-hexadecimal data where hexadecimal data was expected, usually when translating the hexadecimal data to something else.

The methods of the `InvalidBase64Data` exception class are:

**Position** This `readOnly` property is the position in the (expected) hexadecimal string where a non-hexadecimal character was found. `Position` is `0` if the exception was caused because the method object string contained an odd number of characters.

**New** This callable constructor generates an instance of an `InvalidHexData` exception. As shown below, the required argument of the `New` method is a setting of the `Position` property.

To produce an `InvalidHexData` exception, you typically use a User Language Throw statement with an `InvalidHexData` `New` constructor. The `New` method format follows:

```
[%(InvalidHexData):]New(Position=pos)
```

#### **InvalidHexData constructor syntax**

*pos* above is the numeric value to be assigned to the exception object's Position property.

### Example

The following statement throws an InvalidHexData exception with the position set to 1:

```
throw %(invalidHexData):new(position=1)
```

## 5.4.4 InvalidRegex class

The InvalidRegex exception class describes an exception associated with an invalid regular expression being passed to a method that takes a regular expression argument.

The methods of the InvalidRegex exception class are:

**Code** This readOnly property is an integer code that provides some details about the nature of the error. This code is really only useful to someone familiar with the implementation of the regular expression processor, so it is likely to be useful only to Sirius Software technical support.

No assumptions should be made that a particular error will always produce the same code from release to release of the *Sirius Mods*, or even between runs of an Online.

**Description** This readOnly property is a text explanation of the problem. As descriptive text tends to do, this text **might** change between releases of the *Sirius Mods*.

**Position** This readOnly property is the position in the regular expression where it was determined that the regular expression was invalid. It is possible that the mistake was earlier but was not detected until later.

**New** This callable constructor generates an instance of an InvalidRegex exception. As shown below, the required argument of the New method is a setting of the Code, Description, and Position properties.

To produce an InvalidRegex exception, you typically use a User Language Throw statement with an InvalidRegex New constructor. The New method format follows:

<pre>[% (InvalidRegex):]New(Code=code,      -                         Description=desc, -                         Position=pos)</pre>
---

### InvalidRegex constructor syntax

*code* above is the value to be assigned to the exception object's Code property. *desc* is the value to be assigned to the object's Description property. *pos* is the value to be assigned to the object's Position property.

### Example

The following statement throws an InvalidRegex exception with the position set to 13, the code set to 666, and the description indicating Bad Luck:

```
throw %(invalidRegex):new(position=13, code=666, -
                           description='Bad luck')
```

### 5.4.5 NoFreeDaemons class

The NoFreeDaemons exception class indicates that the Daemon class constructor was invoked, but there were no daemon threads available to service the object.

The only method of the NoFreeDaemons exception class is the New constructor. The class has no properties.

**New** This callable constructor generates an instance of an NoFreeDaemons exception.

To produce a NoFreeDaemons exception, you typically use a User Language Throw statement with an NoFreeDaemons New constructor. The New method format follows:

<code>[% (NoFreeDaemons) : ]New</code>
--

#### NoFreeDaemons constructor syntax

### Example

The following statement throws a NoFreeDaemons exception:

```
throw %(noFreeDaemons):new
```

## 5.5 Enhancement methods

When using a class, it is not uncommon to want to perform operations on objects of the class that are *not* one of the standard methods provided by the class. Before *Sirius Mods* version 7.2, there were three ways of addressing this:

1. Add a new method to the class.
2. Create an extension of the class that contains the new method.

3. Create a shared method in another class that takes objects of the first class as input parameters.

*Sirius Mods* version 7.2 introduced a new type of shared method called an **enhancement method** to provide a better way of invoking a method on an object of another class. Because enhancement methods do not operate on objects of the containing class, they are considered shared methods of that class, so they are declared inside the Public Shared or Private Shared blocks of the class.

An enhancement method declaration has the following syntax:

`<method> (<class>):<name> [<otherMethodDesc>]`

### Enhancement method declaration syntax

Where:

- <method>** The method type — Subroutine, Function, or Property. An enhancement method cannot be a constructor.
- <class>** The class of the objects to which the method applies. It cannot be the containing class nor an extension class of the containing class.
- <name>** The name of the enhancement method. The name can be the same name as that of other methods in the class, shared or non-shared, as long as it is not the same name as an enhancement method on the same class.
- <otherMethodDesc>** Method parameters, method type (for Functions and Properties), and other method qualifiers (like AllowNullObject). Any descriptors available to other methods are available to enhancement methods.

An enhancement method invocation has the following syntax:

`<object>:(<+containerClass>)<name> [(<arguments>)]`

### Enhancement method invocation syntax

Where:

- <object>** An object variable of the class against which the enhancement method operates.
- <+containerClass>** The name of the class that contains the enhancement method definition. Note that the class name must be preceded by a plus

sign (+) to indicate an enhancement method invocation. The plus sign distinguishes an enhancement method container class name from other uses of class names inside parentheses that might appear in the same context.

- <name>** The name of the enhancement method.
- <parameters>** Any arguments the enhancement method might take as input. Optional, default, and named parameters work exactly the same way with enhancement methods as with any other methods.

For example, to declare and define an enhancement method version of the EveryOther method described earlier, one would do something like the following:

```
class utility
  public shared
    function (stringlist):everyOther -
      is object stringlist
    end public shared
  function (stringlist):everyOther -
    is object stringlist
    %i      is float
    %outList is object stringlist
    %outlist = new
    for %i from 1 to %this:count by 2
      %outlist:add(%this(%i))
    end for
    return %outlist
  end function
end class
```

As can be seen in this example, an enhancement method has an implicitly declared object variable called `%this`. As with unshared methods, the `%this` variable is a reference to the method object. Unlike unshared methods, the `%this` variable is not an object of the containing class, but is, instead, an instance of the class to which the enhancement method applies.

This enhancement method could then be invoked as follows:

```
%mylist:(+utility)everyOther:print
```

As this example illustrates, the syntax for invoking an enhancement method is more natural from an object-oriented perspective than invoking a shared method that does the same thing.

Using an enhancement method in a chain of methods makes this point even clearer:

```
%mylist:sortNew('1,20,A'):(+utility)everyOther:print
```

This chain of methods can be read from left to right rather than from the inside out.

### 5.5.1 Enhancement methods and inheritance

Enhancement methods are never automatically inherited by extension classes of the methods to which they apply, regardless of whether the `Inherit` keyword is specified on the class declaration for the extension class. For example, the `EveryOther` method in the preceding section cannot be directly applied to `%mylist` if `%mylist` is actually an object of class `Funnylist`, where `FunnyList` is an extension of class `Stringlist`. However, the method can be applied by explicitly specifying the name of the class to which the method applies:

```
%mylist:(stringlist)(+utility)everyOther:print
```

In some languages this is referred to as **casting**, that is, casting a variable as one of its base classes.

You can also explicitly indicate that an extension class is to inherit a base class's enhancement method with a special form of the `Inherit` statement in the `Public` or `Private Shared` blocks:

```
public shared
  function (stringlist):everyOther -
    is object stringlist
  inherit (funnylist):everyOther from stringlist
end public shared
```

If this were specified, the `EveryOther` enhancement method could be applied to objects of class `Funnylist` without any extra qualification.

**Note:** Because an enhancement method is not defined in the class to which the method applies, it cannot reference private variables in that class. However, it **can** reference private members of instances of objects of the containing class.

In fact, one typical application for enhancement methods is to provide a method for creating a new instance of the containing class from an instance of the method object class. These are a special kind of factory method. Such a method might not need to access private members of the method object class, but it might need to access private members of the containing class, so an enhancement factory method is perfectly suitable.

### 5.5.2 Intrinsic Enhancement methods

In addition to Enhancement methods, *Sirius Mods 7.2* introduces Intrinsic methods and classes ([“Intrinsic classes” on page 18](#)). And, just as you can create enhancement methods for other system classes, you can create enhancement methods for intrinsic classes, specifically for the `Float` and `String` classes.

For example, the following definition creates an enhancement method that calculates the length of the hypotenuse of a triangle given the length of one side as the method object and the other side as a parameter:

```
class calc
  public shared
    function (float):hypotenuse(%otherSide is float) -
      is float
    end public shared
  function (float):hypotenuse(%otherSide is float) -
    is float
    return ((%this * %this) + -
      (%otherSide * %otherSide)):squareRoot
  end function
end class
```

You can invoke the above method as follows:

```
%hyp = 3:(+calc)hypotenuse(4)
```

The preceding statement sets `%hyp` to 5. As can be seen in this example, for a Float enhancement method, the implicitly defined `%this` variable has a Float datatype.

The following is an example of a String enhancement method that returns the number of vowels in a string:

```
class myString
  public shared
    function (string):vowels is float
  end public shared
  function (string):vowels is float
    %i      is float
    %vowels is float
    for %i from 1 to %this:length
      if %this:substring(%i, 1): -
        positionIn('aeiouAEIOU') then
        %vowels = %vowels + 1
      end if
    end for
    return %vowels
  end function
end class
```

You can invoke the above method as follows:

```
%nvowels = 'Canberra':(+myString)vowels
```

The preceding statement sets `%nvowels` to 3. For a String enhancement method, the implicitly defined `%this` variable has a Longstring datatype.

### 5.5.3 Enhancement methods for Collections

Enhancement methods can be added to **any** class, including Collection classes (ArrayLists, NamedArrayLists, and so on). Given this capability, it is quite common for a class to have a need to create an enhancement method on a collection of that class.

For example, you might have an Order class that describes an order for widgets. You might want to provide an enhancement method on ArrayList of Order objects that creates a new ArrayList of objects that contains items that need to be back-ordered (there are insufficient widgets in stock to satisfy the order). The method might look something like:

```
class order
  public shared
    function (arraylist of object order):backorderlist -
      is collection arraylist of object order
  end public shared
  function (arraylist of object order):backorderlist -
    is collection arraylist of object order
    %i is float
    %warehouse is object warehouse global
    %newlist is collection arraylist of object order
    %newlist = new
    for %i from 1 to %this:count
      if %this(%i):number gt -
        %warehouse:instock(%this(%i):widgetId)
        %newlist:add(%this)
      end if
    end for
    return %newlist
  end function
end class
```

To invoke this enhancement method, one might do something like:

```
%backOrders = %orders:(+order)backOrderlist
```

However, because a class has a special relationship to collections of objects of that class, it is not necessary to indicate the class of the enhancement method. That is, you can also write the above method invocation as:

```
%backOrders = %orders:backOrderlist
```

It is possible to create enhancement methods with the same name as standard collection methods. For example, it is possible to create an Add method. If you create such a method, the collection class method is completely hidden. That is, there is no way to invoke the collection class method of the same name, even from inside the enhancement method. For this reason, it is generally not a good idea to create an enhancement method for a collection class with the same name as a collection class method.

The ability to hide collection class methods is available mainly to preserve backward compatibility when a new collection class method is introduced. If there were already enhancement methods of the same name, those methods would continue to be invoked.

## 5.6 **ReadOnly and Protected variables**

In *Sirius Mods 7.2* and later, it is possible to declare a class variable as `ReadOnly` or `Protected`. A `ReadOnly` variable can be examined by code outside the class, but it can only be updated inside the class. A `Protected` variable can be examined by code outside the class, but it can only be updated inside the class or by code inside an extension class. Since a `ReadOnly` or `Protected` variable must be accessible outside the class, it would make no sense for such a variable to be in a `Private` section, so this is not allowed.

`ReadOnly` and `Protected` variables are useful for providing an efficient means of accessing relatively static information about objects in a class. Unlike a property, retrieving `ReadOnly` or `Protected` variables requires no code to be run in the class.

The term *Protected* is something of a misnomer. In a very real sense, `Protected` variables aren't protected, at all. After all, they can be updated by extension classes. Their real purpose is to act as a sort of overridable variable, that is, a value that can be overridden by an extension class.

For example, suppose a class has a `ReadOnly` variable called `PartNumber`. Perhaps `PartNumber` is set by the class's constructor, and then it is never set again (for a particular object instance). Now, suppose this class wants to allow extension classes to set a different `PartNumber`, probably in their constructors. One approach would be to make `PartNumber` an `Overridable ReadOnly` property. But this is a somewhat heavyweight approach, as it requires a bit of code (the property `Get` method) for each extension class, and this code has to be run every time `PartNumber` is retrieved. Instead, `PartNumber` could be made a `Protected` variable, and an extension class's constructor could simply set it after calling the base class constructor. This allows the extension classes to override the base class's `PartNumber` using little extra code and using a very efficient access path for the value.

`Protected` variables differ from `Overridable ReadOnly` properties, however, in that an `Overridable ReadOnly` property always guarantees that the extension class's code is run, so it overrides the base class's properties. With a `Protected` variable, it would be possible for an extension class to set it, and for the base class to then set it to something else, undoing the extension class action.

Because they can be updated by both base and extension classes, `Protected` variables are probably most useful for very static values, like values that are only set by the constructors. Use of `Protected` variables that can be set throughout the life of objects of a class is likely to be error-prone, as it requires careful coordination of updates between the base and extension classes.

## 5.7 New System class methods

These new methods are added to the System class: Arguments, CallStack, CallStackDepth, and CallStackCaller.

### 5.7.1 The Arguments method

The Arguments method allows evaluation-time examination of INCLUDE arguments. This shared function returns the arguments passed in the INCLUDE command for the procedure that contained the Begin for the current request.

```
%args = %(system):Arguments
```

#### Arguments syntax

For example, if procedure FOO contained the following:

```
b
printText {~} = '{%(system):arguments}'
end
```

The following requests (preceded by >) would produce the following outputs:

```
> I F00 This is a test
%(system):arguments = ' This is a test'

> I F00,This is a test
%(system):arguments = ',This is a test'

> I F00          This is a test
%(system):arguments = '          This is a test'

> I F00 This,is,a,test
%(system):arguments = ' This,is,a,test'

> I F00
%(system):arguments = ''
```

Note that everything after the INCLUDEd procedure name is returned by the Arguments method, including any blanks or commas after the procedure name.

### 5.7.2 The CallStack method

This shared function returns a Stringlist containing information about the current call stack: information about the caller of the current method or subroutine, the caller of that caller, and so on.

```
%callList = %(system):CallStack
```

#### CallStack syntax

The CallStack method returns the name of the procedure and the line within the procedure that made the calls of the current method or subroutine. Generally, each returned StringList item has the name of the file containing the calling procedure at positions one through eight, followed by a blank, followed by the name of the calling procedure, followed by a blank, followed by the line number within the calling procedure.

### 5.7.3 The CallStackCaller method

This shared function returns a string containing information about the caller of the current method or subroutine. The information can extend back to a specific number of call levels.

```
%callString = %(system):CallStackCaller(%depth)
```

#### CallStackCaller syntax

The CallStackCaller method returns the name of the procedure and the line within the procedure that made the calls of the current method or subroutine. Generally, the returned string has the name of the file containing the calling procedure at positions one through eight, followed by a blank, followed by the name of the calling procedure, followed by a blank, followed by the line number within the calling procedure.

### 5.7.4 The CallStackDepth method

This shared function returns a number indicating the depth of the call stack.

```
%depth = %(system):CallStackDepth
```

#### CallStackDepth syntax

At level-0 (immediately inside a Begin/End), CallStackDepth always returns 0. For a method or subroutine called directly from level-0 code, CallStackDepth would return 1. For a method called from a method called from level-0, CallStackDepth would return 2; and so on.

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

## 6.1 Multiple XPath predicates

XPath expressions may now contain multiple predicates in a single step. Previous versions of *Janus SOAP* supported some, but not all, cases of multiple predicates.

The primary purpose of the new multiple predicate support is to let you use the `position()` function to filter based on the position of nodes from the preceding *predicate*, rather than from the step's *nodetest*.

For example, the following expression selects the second `chapter` child of the `book`, if its author is Alex:

```
/book/chapter[author="Alex" and 2]
```

While the following expression selects the second `chapter` child that is authored by Alex:

```
/book/chapter[author="Alex"] [2]
```

## 6.2 not() function in XPath predicates

The `not()` function is now supported in XPath predicates. The argument is a Boolean expression, and the result is `true` if the value of the argument is `false`, and `false` otherwise.

**Note:** The result of the `not()` function applied to a comparison expression is different than the same expression with the complementary comparison. For example, this statement selects children that have the value of the status attribute equal to "pending":

```
%lis = %nod:SelectNodes('*[@status="pending"]')
```

The following statement selects children that have the value of the status attribute equal to something other than "pending":

```
%lis = %nod:SelectNodes('*[@status!="pending"]')
```

And the following statement selects children that have the value of the status attribute equal to something other than "pending", or that have no status attribute:

```
%lis = %nod:SelectNodes('[not(@status="pending")]')
```

### 6.3 Nested XPath predicates

An XPath expression is now allowed to have a predicate that contains a location path which itself contains a predicate; that is, predicates may be nested.

For example, this selects Chapters whose first Section has a *Racy* attribute:

```
%lis = %bk:SelectNodes('Chapter[Section[1 and @Racy]]')
```

### 6.4 Boolean parentheses in XPath predicates

In an XPath expression that has a predicate that contains a Boolean expression, parentheses are now allowed for grouping in the Boolean operands. For example, prior to this release, the following predicate was not supported in *Janus SOAP*:

```
chapter[@type="methods" and  
(@class="Stringlist" or @class="Daemon")]
```

### 6.5 URI now allows hexadecimal escape (%hh)

The various places in *Janus SOAP* that process a URI (for example, the third argument of the *AddElement* method, or the string input to the *LoadXml* method) now allow a URI to contain a "hexadecimal escape," that is, a percent sign (%) followed by two hexadecimal digits (which may be either uppercase or lowercase when the letters A-F are used).

Note that there is no replacement of the hexadecimal values when URI processing is performed. For example, even though the ASCII code for the number "4" is hexadecimal 34, the following two URIs are different and distinct:

```
http://my.URI.number4  
http://my.URI.number%34
```

Thus, for instance, the following fragment:

```
%n = %d:AddElement('x', , 'http://my.URI.number4')
      %n:AddElement('x', , 'http://my.URI.number%34')
%d:Print
%d:SelectionPrefix('f') = 'http://my.URI.number4'
Print %d:SelectCount('//f:x') And 'matching node(s)'
```

Will have the following result:

```
<x xmlns="http://my.URI.number4">
  <x xmlns="http://my.URI.number%34"/>
</x>
1 matching node(s)
```

**Note:**

- In addition to this feature, the error messages issued when an invalid URI is encountered have been improved to better explain the specific problem with the URI.
- See [“Additional URI restrictions” on page 61](#) for some other notes regarding changes to URI processing.
- A less restrictive version of this feature was provided in version 7.1 of the *Sirius Mods* by a maintenance zap. However, as mentioned in [“Various URIs now recognized to be invalid” on page 62](#), since it is less restrictive, the zap provides an opportunity for a form of upward incompatibility (if you use a percent sign that is not followed by two valid hexadecimal digits).



---

**CHAPTER 7** *Janus Sockets*

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

## 7.1 HTTPRequest class

### 7.1.1 Send method

Prior to the addition of the Send method, the HTTPRequest class had methods only for HTTP GETs and POSTs. The new Send method is a way to carry out HTTP methods in addition to GET or POST.

The Send function sends an HTTP request to an HTTP server using a parameter value to identify which of any of the HTTP method types (GET, POST, PUT, and so on) you want the function to perform. Any HTTP method type is valid as long as its name is 16 characters or less.

The HTTP method type is specified with the Method (name required) parameter. If you set the Method value to `GET` or `POST`, the Send method invocation becomes equivalent to the existing Get method or Post method, respectively, of the HTTPRequest class.

```
%httpresp = %httpreq:Send([[Port=] name,]      -  
                        [[Cancel=] num,]      -  
                        Method=methodname )
```

#### Send syntax



The following features are new or changed in *Janus Web Server*.

## 8.1 Support for additional HTTP method types

*Janus Web Server* now accepts any HTTP method in a request as long as the method name is 16 bytes or shorter. This feature is provided by:

- Adding support for specifying additional HTTP method types in Janus Web rules.

The new `OTHER` option in the *method* parameter in the JANUS WEB command refers to all methods other than PUT, GET, POST, and HEAD. A method whose name exceeds 16 characters is not allowed.

Examples of rules that use OTHER follow:

```
JANUS WEB WEBPORT ALLOW OTHER * USER *
JANUS WEB WEBPORT ON    OTHER /WEBDAV/* OPEN FILE WEBDAV
CMD 'I *'
```

- The ANY option of the *method* parameter now means any HTTP method whose name is 16 bytes or shorter. Prior to *Sirius Mods* version 7.2, ANY referred only to PUT, GET, POST, or HEAD.

**Note:** This is a slight backward incompatibility: if a web rule specifies ANY, URLs indicating methods other than GET, POST, PUT, or HEAD may now trigger unexpected Web Server actions.

You can detect the method type in an application by a `$web_hdr_parm('METHOD')` query.

- Adding the new Send method in the HTTPRequest class ([“Send method” on page 55](#)).



The following are new or changed features in the *Sirius Functions*:

## 9.1 **NOURGMSG option of \$Wakeup**

Prior to *Sirius Mods 7.2*, the waiting time that remains for a user paused by a \$Wakeup call is cancelled if a BROADCAST URGENT message is sent to the user. As of *Sirius Mods 7.2*, the \$Wakeup function accepts an optional argument so that BROADCAST URGENT does **not** cause the \$Wakeup function time to complete.

The string **NOURGMSG** (or any upper/lower case variant) is the optional argument for \$Wakeup that prevents a paused user from being immediately wakened by BROADCAST URGENT.

To change \$Wakeup in *Sirius Mods* version 6.7 or 7.1 so that a BROADCAST URGENT does not cause \$Wakeup to complete (**without** any control of an argument), custom zaps are provided in the *Sirius Mods* release notes for those versions.

## 9.2 **\$Lstr\_Word return word longer than 255 bytes**

The \$Lstr\_Word function now may return a Longstring, whose length exceeds 255 bytes. Previously, attempting to return such a result either caused request cancellation, or caused truncation to 255 bytes of the returned string.

This change was actually introduced by maintenance as ZAP7236, and it was also introduced as maintenance in version 7.1 of the *Sirius Mods*



---

**CHAPTER 10** *Compatibility/Bug fixes*

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 immediately prior version (7.1).

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).

## **10.1 Backwards incompatibilities**

Backwards incompatibilities are described per product in the following sections.

### **10.1.1 Janus SOAP Xml API**

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

#### **10.1.1.1 Additional URI restrictions**

As discussed in “[Various URIs now recognized to be invalid](#)” on page 62, in older versions of *Janus SOAP* the handling of a URI by various methods allowed forms of URI which are not legal according to the standard (RFC2396).

These URI are no longer allowed; they are parse errors (and so handled by the ErrRet option) for deserialization methods, and they are request cancellation errors for other methods that have a URI argument.

#### **10.1.2 Janus Web Server**

The following backwards compatibility issues have been introduced in the *Janus Web Server*.

### 10.1.2.1 Support for additional HTTP method types in Janus Web rules

As described in “[Support for additional HTTP method types](#)” on page 57, the ANY option of the *method* parameter in the JANUS WEB command now means any HTTP method whose name is 16 bytes or shorter. Prior to *Sirius Mods* version 7.2, ANY referred only to PUT, GET, POST, or HEAD.

For Janus Web port rules using the ANY option, this backward incompatibility now allows URLs specifying methods other than GET, POST, PUT, or HEAD to trigger an unexpected Web Server action.

## 10.2 Fixes in Sirius Mods 7.2 but not in 7.1

This section lists fixes to functionality existing in the *Sirius Mods* version 7.1 but which, due to the absence of customer problems, have not, as of the date of the release, been fixed in that version.

### 10.2.1 Various URIs now recognized to be invalid

Previously, some illegal URI strings were allowed for various XML deserialization methods, and they were allowed as the URI argument of various methods in the *XmlDoc* and *XmlNode* classes. These are some cases that previously were allowed but no longer are:

- A URI “fragment” (which starts with the # character) immediately after the first colon in a URI. For example, the following was allowed:

```
%doc:LoadXml('<x xmlns="http:#rrr"/>')
```

- A URI that starts with a decimal digit. For example, the following was allowed:

```
%doc:LoadXml('<x xmlns="10:20"/>')
```

- A URI that starts with a colon. For example, the following was allowed:

```
%doc:LoadXml('<x xmlns=":bad"/>')
```

- With a maintenance zap to version 7.1 of the *Sirius Mods*, it was valid to have a percent sign (%) that is not followed by two valid hexadecimal digits. For example, the following was allowed:

```
%doc:LoadXml('<x xmlns="http://zero%3"/>')
```

As of version 7.2, each of the above URIs results in a parsing error. Similarly, a request cancellation results if you use an invalid URI in other methods, such as:

```
%node:AddElement('someName', 'http:#rrr"/>')
```

As mentioned in “[Additional URI restrictions](#)” on page 61, these represent a small upwards incompatibility.

**Note:** In addition, in previous versions of *Janus SOAP*, some invalid forms of URI were recognized as invalid by deserialization methods, but they were not recognized to be invalid by other methods.

## 10.2.2 Processing certain alternatives in Regex expressions

In versions of the *Sirius Mods* prior to 7.2, Sirius regex fails to correctly process a subexpression in which one alternative is a proper initial substring of a later alternative.

Sirius regex prior to 7.1 erroneously fails to backtrack and try the second alternative in a case like the following regular expression (broken across lines for readability) for finding dates like "27February2007" or "1 Jan 08":

```
(\d{1,2})[ ]{0,1}
(Jan|January|Feb|February|Mar|March)
[ ]{0,1}(\d{2,4})
```

where

**(\d{1,2})** Matches one or two digits.

**[ ]{0,1}** Matches zero or one blank (could also be written “ ?”).

**(Jan|January|Feb|February|Mar|March)**

Matches the names of the first three months, either short or long forms.

**[ ]{0,1}** Matches zero or one blank.

**(\d{2,4})** Matches two to four digits.

In *Sirius Mods* 7.1, the regex does not work correctly when the short forms of the month names are given before the long forms. If the string to be matched is something like “blah blah blah 27 January 2008 yada yada yada”, the first alternative of the regex matches as far as it can in the target string, that is, up through the “27 Jan” portion, and it fails (appropriately) against “uary 2008 ...”. The processing should then backtrack and try the next alternative in the regex, *January*. However, *Sirius Mods* 7.1 processing does **not** retry the next alternative, so it fails (inappropriately) to match this target string.

In *Sirius Mods* 7.2, the regex processing does backtrack and retry properly the later alternative, and it reports a successful match against the target string.

Unfortunately, the code changes for this fix are too extensive to be rolled into a ZAP for *Sirius Mods* 7.1, but there is a workaround for 7.1 users: in any regex string of such alternates, place longer strings ahead of their corresponding shorter initial substrings.

Thus, the preceding example does not fail if the string is like:

... (January | Jan | February | Feb | Mar | March) ...

**Note:** There is no error if the first alternative is other than a proper initial substring of the second: A regex like “Jan|Feb|MyAuntSally|YourAuntSally” works properly.

### 10.3 Version corequisites

This section lists any restrictions on usage of various products (including *Sirius Mods* itself) that will be imposed by use of version 7.2 of *Sirius Mods*.

- There are no corequisites associated with *Sirius Mods* 7.2.