Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2016 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

For details about other licensing agreements for third-party products, you must read the JADE ReadMe.txt file.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Before You Begin</strong></td>
<td>xi</td>
</tr>
<tr>
<td>Who Should Read this Encyclopaedia</td>
<td>xi</td>
</tr>
<tr>
<td>What's Included in this Encyclopaedia</td>
<td>xi</td>
</tr>
<tr>
<td>Related Documentation</td>
<td>xi</td>
</tr>
<tr>
<td>Conventions</td>
<td>xii</td>
</tr>
<tr>
<td><strong>Chapter 1  Primitive Types</strong></td>
<td>13</td>
</tr>
<tr>
<td>Overview</td>
<td>14</td>
</tr>
<tr>
<td>Any Type</td>
<td>15</td>
</tr>
<tr>
<td>Any Methods</td>
<td>15</td>
</tr>
<tr>
<td>asString</td>
<td>16</td>
</tr>
<tr>
<td>display</td>
<td>16</td>
</tr>
<tr>
<td>getName</td>
<td>16</td>
</tr>
<tr>
<td>isKindOf</td>
<td>16</td>
</tr>
<tr>
<td>Binary Type</td>
<td>18</td>
</tr>
<tr>
<td>Binary Constants</td>
<td>18</td>
</tr>
<tr>
<td>Binary Methods</td>
<td>19</td>
</tr>
<tr>
<td>ansiToString</td>
<td>20</td>
</tr>
<tr>
<td>ansiToUnicode</td>
<td>20</td>
</tr>
<tr>
<td>asDecimal</td>
<td>20</td>
</tr>
<tr>
<td>asGuidString</td>
<td>21</td>
</tr>
<tr>
<td>base64Encode</td>
<td>21</td>
</tr>
<tr>
<td>base64EncodeNoCrLf</td>
<td>22</td>
</tr>
<tr>
<td>bufferAddress</td>
<td>22</td>
</tr>
<tr>
<td>bufferMemoryAddress</td>
<td>23</td>
</tr>
<tr>
<td>compressToBinary</td>
<td>24</td>
</tr>
<tr>
<td>convertPicture</td>
<td>25</td>
</tr>
<tr>
<td>convertToFile</td>
<td>25</td>
</tr>
<tr>
<td>display</td>
<td>26</td>
</tr>
<tr>
<td>fromANSIToString</td>
<td>26</td>
</tr>
<tr>
<td>fromANSIToStringUtf8</td>
<td>26</td>
</tr>
<tr>
<td>length</td>
<td>26</td>
</tr>
<tr>
<td>maxLength</td>
<td>26</td>
</tr>
<tr>
<td>pictureSize</td>
<td>27</td>
</tr>
<tr>
<td>pictureType</td>
<td>27</td>
</tr>
<tr>
<td>posBinary</td>
<td>27</td>
</tr>
<tr>
<td>posByte</td>
<td>27</td>
</tr>
<tr>
<td>unpackCString</td>
<td>26</td>
</tr>
<tr>
<td>uuidAsString</td>
<td>29</td>
</tr>
<tr>
<td><strong>Boolean Type</strong></td>
<td>31</td>
</tr>
<tr>
<td>Boolean Method</td>
<td>31</td>
</tr>
<tr>
<td>display</td>
<td>32</td>
</tr>
<tr>
<td><strong>Byte Type</strong></td>
<td>33</td>
</tr>
<tr>
<td>Using Byte Types in Assignments</td>
<td>33</td>
</tr>
<tr>
<td>Byte Methods</td>
<td>34</td>
</tr>
<tr>
<td>bitAnd</td>
<td>35</td>
</tr>
<tr>
<td>bitNot</td>
<td>35</td>
</tr>
<tr>
<td>bitOr</td>
<td>35</td>
</tr>
<tr>
<td>bitXor</td>
<td>36</td>
</tr>
<tr>
<td>display</td>
<td>36</td>
</tr>
<tr>
<td>isEven</td>
<td>36</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Character Type</th>
<th>Character Methods</th>
<th>Date Type</th>
<th>Historical Note about the Date Type</th>
<th>Date Primitive Type Examples</th>
<th>Date Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Character Type

- `isOdd` (37)
- `max` (37)
- `min` (37)
- `numberFormat` (37)
- `padLeadingWith` (38)
- `parseCurrencyWithCurrentLocale` (38)
- `parseCurrencyWithFmtAndLcid` (39)
- `parseNumberWithCurrentLocale` (39)
- `parseNumberWithFmtAndLcid` (40)
- `userCurrencyFormat` (40)
- `userCurrencyFormatAndLcid` (41)
- `userNumberFormat` (41)
- `userNumberFormatAndLcid` (42)

### Character Methods

- `compareEql` (43)
- `compareGeneric` (44)
- `compareGeq` (45)
- `compareGtr` (46)
- `compareLeq` (47)
- `compareLss` (48)
- `compareNeq` (49)
- `display` (50)
- `isAlpha` (51)
- `isDelimiter` (51)
- `isHex` (52)
- `isLower` (52)
- `isNumeric` (52)
- `isPrintable` (53)
- `isUpper` (53)
- `makeString` (53)
- `setByteOrderLocal` (54)
- `setByteOrderRemote` (54)
- `toHex` (55)
- `toLower` (55)
- `toUpper` (55)

### Date Type

- `day` (56)
- `dayName` (59)
- `dayNameWithLcid` (60)
- `dayOfWeek` (60)
- `dayOfYear` (60)
- `display` (61)
- `format` (61)
- `isFormatable` (62)
- `isLeapYear` (62)
- `isValid` (63)
- `longFormat` (63)
- `month` (63)
- `monthName` (64)
- `monthNameWithLcid` (64)
- `parseForCurrentLocale` (65)
- `parseLongWithCurrentLocale` (65)
- `parseLongWithFmtAndLcid` (66)
- `parseLongWithPicAndLcid` (66)
- `parseShortWithCurrentLocale` (68)
- `parseShortWithFmtAndLcid` (68)
**Contents**

<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>parseShortWithPicAndLcid</td>
<td>69</td>
</tr>
<tr>
<td>setByteOrderLocal</td>
<td>70</td>
</tr>
<tr>
<td>setByteOrderRemote</td>
<td>71</td>
</tr>
<tr>
<td>setDate</td>
<td>71</td>
</tr>
<tr>
<td>setDateYearAbsolute</td>
<td>72</td>
</tr>
<tr>
<td>shortDayNameWithLcid</td>
<td>72</td>
</tr>
<tr>
<td>shortFormat</td>
<td>72</td>
</tr>
<tr>
<td>shortMonthNameWithLcid</td>
<td>73</td>
</tr>
<tr>
<td>subtract</td>
<td>73</td>
</tr>
<tr>
<td>userFormat</td>
<td>74</td>
</tr>
<tr>
<td>userLongFormatAndLcid</td>
<td>74</td>
</tr>
<tr>
<td>userLongFormatPicAndLcid</td>
<td>74</td>
</tr>
<tr>
<td>userShortFormatAndLcid</td>
<td>75</td>
</tr>
<tr>
<td>userShortFormatPicAndLcid</td>
<td>75</td>
</tr>
<tr>
<td>year</td>
<td>76</td>
</tr>
<tr>
<td><strong>Decimal Type</strong></td>
<td>77</td>
</tr>
<tr>
<td><strong>Decimal Methods</strong></td>
<td>77</td>
</tr>
<tr>
<td>abs</td>
<td>78</td>
</tr>
<tr>
<td>asBinary</td>
<td>78</td>
</tr>
<tr>
<td>asDecimal</td>
<td>79</td>
</tr>
<tr>
<td>currencyFormat</td>
<td>79</td>
</tr>
<tr>
<td>display</td>
<td>79</td>
</tr>
<tr>
<td>getDeclaredPrecision</td>
<td>80</td>
</tr>
<tr>
<td>getDeclaredScaleFactor</td>
<td>80</td>
</tr>
<tr>
<td>numberFormat</td>
<td>80</td>
</tr>
<tr>
<td>parseCurrencyWithCurrentLocale</td>
<td>80</td>
</tr>
<tr>
<td>parseCurrencyWithFmtAndLcid</td>
<td>81</td>
</tr>
<tr>
<td>parseNumberWithCurrentLocale</td>
<td>82</td>
</tr>
<tr>
<td>parseNumberWithFmtAndLcid</td>
<td>82</td>
</tr>
<tr>
<td>rounded</td>
<td>82</td>
</tr>
<tr>
<td>rounded64</td>
<td>83</td>
</tr>
<tr>
<td>roundedTo</td>
<td>83</td>
</tr>
<tr>
<td>setByteOrderLocal</td>
<td>84</td>
</tr>
<tr>
<td>setByteOrderRemote</td>
<td>84</td>
</tr>
<tr>
<td>truncated</td>
<td>85</td>
</tr>
<tr>
<td>truncated64</td>
<td>85</td>
</tr>
<tr>
<td>truncatedTo</td>
<td>86</td>
</tr>
<tr>
<td>userCurrencyFormat</td>
<td>86</td>
</tr>
<tr>
<td>userCurrencyFormatAndLcid</td>
<td>87</td>
</tr>
<tr>
<td>userNumberFormat</td>
<td>87</td>
</tr>
<tr>
<td>userNumberFormatAndLcid</td>
<td>87</td>
</tr>
<tr>
<td><strong>Integer Type</strong></td>
<td>89</td>
</tr>
<tr>
<td><strong>Integer Methods</strong></td>
<td>90</td>
</tr>
<tr>
<td>abs</td>
<td>91</td>
</tr>
<tr>
<td>bitAnd</td>
<td>91</td>
</tr>
<tr>
<td>bitNot</td>
<td>91</td>
</tr>
<tr>
<td>bitOr</td>
<td>92</td>
</tr>
<tr>
<td>bitXor</td>
<td>92</td>
</tr>
<tr>
<td>display</td>
<td>93</td>
</tr>
<tr>
<td>isEven</td>
<td>93</td>
</tr>
<tr>
<td>isOdd</td>
<td>93</td>
</tr>
<tr>
<td>max</td>
<td>93</td>
</tr>
<tr>
<td>min</td>
<td>94</td>
</tr>
<tr>
<td>numberFormat</td>
<td>94</td>
</tr>
<tr>
<td>padLeadingWith</td>
<td>94</td>
</tr>
<tr>
<td>parseCurrencyWithCurrentLocale</td>
<td>95</td>
</tr>
<tr>
<td>parseCurrencyWithFmtAndLcid</td>
<td>95</td>
</tr>
<tr>
<td>parseNumberWithCurrentLocale</td>
<td>96</td>
</tr>
<tr>
<td>parseNumberWithFmtAndLcid</td>
<td>96</td>
</tr>
<tr>
<td>setByteOrderLocal</td>
<td>97</td>
</tr>
</tbody>
</table>
Contents

log ................................................................. 119
log10 ............................................................... 119
max ................................................................. 119
min ................................................................. 120
nan ................................................................. 120
numberFormat .................................................... 120
parseCurrencyWithCurrentLocale ............................... 121
parseCurrencyWithFmtAndLcid ................................ 121
parseNumberWithCurrentLocale ................................ 122
parseNumberWithFmtAndLcid ................................... 122
rounded ............................................................. 123
rounded64 .......................................................... 123
roundedTo .......................................................... 123
roundedUp ........................................................... 124
roundedUp64 ......................................................... 124
setByteOrderLocal ............................................... 125
setByteOrderRemote .............................................. 125
setFloatingPointClassification ................................. 126
sin ................................................................. 126
sqrt ................................................................. 127
tan ................................................................. 127
truncated ............................................................ 127
truncated64 ........................................................ 128
truncatedTo ........................................................ 128
userCurrencyFormat .............................................. 128
userCurrencyFormatAndLcid .................................... 128
userNumberFormat ................................................ 129
userNumberFormatAndLcid ...................................... 129

String Type ....................................................... 130

String Methods .................................................. 131
asANSI .............................................................. 132
asDate .............................................................. 133
asGuid .............................................................. 133
asObject ........................................................... 134
asOid ............................................................... 134
asStringUtf8 ....................................................... 134
asUuid .............................................................. 135
base64Decode ...................................................... 135
bufferAddress ..................................................... 135
bufferMemoryAddress .......................................... 136
compareEqI ........................................................ 137
compareGeneric ................................................ 138
compareGeq ....................................................... 139
compareGtr ....................................................... 140
compareLeq ....................................................... 141
compareLss ....................................................... 142
compareNeq ....................................................... 143
compressToBinary ............................................... 144
display ............................................................. 144
fillString .......................................................... 144
firstCharToLower ............................................... 145
firstCharToUpper ............................................... 145
getHugeTokens .................................................. 145
getNextToken .................................................... 145
getTokens ......................................................... 146
isByte ............................................................. 146
isDecimal ......................................................... 147
isInteger ........................................................ 147
isInteger64 ....................................................... 148
isReal ............................................................. 148
## Contents

<table>
<thead>
<tr>
<th>Category</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>JadeDynamicObjectType Category</td>
<td>202</td>
</tr>
<tr>
<td>JadeErrorCodesDatabase Category</td>
<td>202</td>
</tr>
<tr>
<td>JadeErrorCodesIDE Category</td>
<td>203</td>
</tr>
<tr>
<td>JadeErrorCodesRPS Category</td>
<td>203</td>
</tr>
<tr>
<td>JadeErrorCodesSDS Category</td>
<td>203</td>
</tr>
<tr>
<td>JadeErrorCodesWebService Category</td>
<td>204</td>
</tr>
<tr>
<td>JadeLocaleIdNumbers Category</td>
<td>204</td>
</tr>
<tr>
<td>JadeOdbcCategory</td>
<td>207</td>
</tr>
<tr>
<td>JadeProcessEventsCategory</td>
<td>207</td>
</tr>
<tr>
<td>JadeProfileString Category</td>
<td>208</td>
</tr>
<tr>
<td>KeyCharacterCodes Category</td>
<td>208</td>
</tr>
<tr>
<td>LockDurations Category</td>
<td>210</td>
</tr>
<tr>
<td>LockTimeouts Category</td>
<td>210</td>
</tr>
<tr>
<td>Locks Category</td>
<td>211</td>
</tr>
<tr>
<td>MessageBox Category</td>
<td>211</td>
</tr>
<tr>
<td>NotificationResponses Category</td>
<td>212</td>
</tr>
<tr>
<td>ObjectVolatility Category</td>
<td>212</td>
</tr>
<tr>
<td>PossibleTransientLeaks Category</td>
<td>213</td>
</tr>
<tr>
<td>Printer Category</td>
<td>213</td>
</tr>
<tr>
<td>RPSTransitionHaltCode Category</td>
<td>216</td>
</tr>
<tr>
<td>SDSConnectionStateCategory</td>
<td>217</td>
</tr>
<tr>
<td>SDSDatabaseRolesCategory</td>
<td>217</td>
</tr>
<tr>
<td>SDSEventTypes Category</td>
<td>217</td>
</tr>
<tr>
<td>SDSEorgStateCategory</td>
<td>218</td>
</tr>
<tr>
<td>SDSSecondaryStateCategory</td>
<td>218</td>
</tr>
<tr>
<td>SDSStopTrackingCodesCategory</td>
<td>218</td>
</tr>
<tr>
<td>SDSTakeoverStateCategory</td>
<td>219</td>
</tr>
<tr>
<td>SDSTransactionStatesCategory</td>
<td>219</td>
</tr>
<tr>
<td>SQL Category</td>
<td>220</td>
</tr>
<tr>
<td>Sounds Category</td>
<td>222</td>
</tr>
<tr>
<td>SystemEventsCategory</td>
<td>222</td>
</tr>
<tr>
<td>SystemLimitsCategory</td>
<td>222</td>
</tr>
<tr>
<td>TimerDurationsCategory</td>
<td>223</td>
</tr>
<tr>
<td>UUIDVariantsCategory</td>
<td>223</td>
</tr>
<tr>
<td>UserEventsCategory</td>
<td>223</td>
</tr>
</tbody>
</table>
Before You Begin

The JADE Encyclopaedia of Primitive Types is intended as a major source of information when you are developing or maintaining JADE applications.

Who Should Read this Encyclopaedia

The main audience for the JADE Encyclopaedia of Primitive Types is expected to be developers of JADE application software products.

What's Included in this Encyclopaedia

The JADE Encyclopaedia of Primitive Types has one chapter and one appendix.

<table>
<thead>
<tr>
<th>Chapter 1</th>
<th>Gives a reference to primitive types and the methods that they provide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix A</td>
<td>Gives a reference to global constants</td>
</tr>
</tbody>
</table>

Related Documentation

Other documents that are referred to in this encyclopaedia, or that may be helpful, are listed in the following table, with an indication of the JADE operation or tasks to which they relate.

<table>
<thead>
<tr>
<th>Title</th>
<th>Related to…</th>
</tr>
</thead>
<tbody>
<tr>
<td>JADE Database Administration Guide</td>
<td>Administering JADE databases</td>
</tr>
<tr>
<td>JADE Development Environment Administration Guide</td>
<td>Administering JADE development environments</td>
</tr>
<tr>
<td>JADE Development Environment User's Guide</td>
<td>Using the JADE development environment</td>
</tr>
<tr>
<td>JADE Developer's Reference</td>
<td>Developing or maintaining JADE applications</td>
</tr>
<tr>
<td>JADE Encyclopaedia of Classes</td>
<td>System classes (Volumes 1 and 2), Window classes (Volume 3)</td>
</tr>
<tr>
<td>JADE Installation and Configuration Guide</td>
<td>Installing and configuring JADE</td>
</tr>
<tr>
<td>JADE Initialization File Reference</td>
<td>Maintaining JADE initialization file parameter values</td>
</tr>
<tr>
<td>JADE Object Manager Guide</td>
<td>JADE Object Manager administration</td>
</tr>
<tr>
<td>JADE Synchronized Database Service (SDS) Administration Guide</td>
<td>Administering JADE Synchronized Database Services (SDS), including Relational Population Services (RPS)</td>
</tr>
<tr>
<td>JADE Thin Client Guide</td>
<td>Administering JADE thin client environments</td>
</tr>
</tbody>
</table>
# Conventions

The *JADE Encyclopaedia of Primitive Types* uses consistent typographic conventions throughout.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrow bullet (⇒)</td>
<td>Step-by-step procedures. You can complete procedural instructions by using either the mouse or the keyboard.</td>
</tr>
<tr>
<td><strong>Bold</strong></td>
<td>Items that must be typed exactly as shown. For example, if instructed to type <code>foreach</code>, type all the bold characters exactly as they are printed. File, class, primitive type, method, and property names, menu commands, and dialog controls are also shown in bold type, as well as literal values stored, tested for, and sent by JADE instructions.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Parameter values or placeholders for information that must be provided; for example, if instructed to enter <code>class-name</code>, type the actual name of the class instead of the word or words shown in italic type. Italic type also signals a new term. An explanation accompanies the italicized type. Document titles and status and error messages are also shown in italic type.</td>
</tr>
<tr>
<td><strong>Blue text</strong></td>
<td>Enables you to click anywhere on the cross-reference text (the cursor symbol changes from an open hand to a hand with the index finger extended) to take you straight to that topic. For example, click on the &quot;IsKindOf&quot; cross-reference to display that topic.</td>
</tr>
<tr>
<td>Bracket symbols ([ ] )</td>
<td>Indicate optional items.</td>
</tr>
<tr>
<td>Vertical bar (</td>
<td>)</td>
</tr>
<tr>
<td>Monospaced font</td>
<td>Syntax, code examples, and error and status message text.</td>
</tr>
<tr>
<td><strong>ALL CAPITALS</strong></td>
<td>Directory names, commands, and acronyms.</td>
</tr>
<tr>
<td><strong>SMALL CAPITALS</strong></td>
<td>Keyboard keys.</td>
</tr>
</tbody>
</table>

Key combinations and key sequences appear as follows.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY1+KEY2</td>
<td>Press and hold down the first key and then press the second key. For example, &quot;press SHIFT+F2&quot; means to press and hold down the SHIFT key and press the F2 key. Then release both keys.</td>
</tr>
<tr>
<td>KEY1,KEY2</td>
<td>Press and release the first key, then press and release the second key. For example, &quot;press ALT+F,X&quot; means to hold down the ALT key, press the F key, and then release both keys before pressing and releasing the X key.</td>
</tr>
</tbody>
</table>

In this document, the term Microsoft Windows refers to Windows 10, Windows 8, Windows 7, Windows Server 2012, Windows Server 2008, Windows Vista, or Windows Mobile. When there are differences between the versions of Microsoft Windows, the specific version of Microsoft Windows is stated.

With the exception of the *jade.exe* program, when referring to program executables in this document, the `.exe` file suffix is omitted; for example, *jadclient* refers to *jadclient.exe*. Similarly, the `.dll` (Dynamic Link Library) suffix is omitted. For example, *jomos* refers to *jomos.dll*. 
Chapter 1

Primitive Types

This chapter covers the following topics.

- Overview
- Any Type
- Binary Type
- Boolean Type
- Byte Type
- Character Type
- Date Type
- Decimal Type
- Integer Type
- Integer64 Type
- MemoryAddress Type
- Point Type
- Real Type
- String Type
- StringUtf8 Type
- Time Type
- TimeStamp Type
- TimeStampInterval Type
- TimeStampOffset Type
Overview

The type of a method or property determines the range of values that the method or property can take and its interface (or protocol). A type can be a primitive type, a class, or a JADE interface.

The primitive types are summarized in the following table.

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>Represents any object reference or primitive value</td>
</tr>
<tr>
<td>Binary</td>
<td>Represents binary data</td>
</tr>
<tr>
<td>Boolean</td>
<td>Contains Boolean value <strong>true</strong> or <strong>false</strong></td>
</tr>
<tr>
<td>Byte</td>
<td>A single byte unsigned value (8 bits)</td>
</tr>
<tr>
<td>Character</td>
<td>Any single ANSI or Unicode character</td>
</tr>
<tr>
<td>Date</td>
<td>Julian day number</td>
</tr>
<tr>
<td>Decimal</td>
<td>Number with specific decimal format</td>
</tr>
<tr>
<td>Integer</td>
<td>Signed 32-bit integer (whole number)</td>
</tr>
<tr>
<td>Integer64</td>
<td>Signed 64-bit integer (whole number)</td>
</tr>
<tr>
<td>MemoryAddress</td>
<td>Represents a memory address</td>
</tr>
<tr>
<td>Point</td>
<td>Represents x and y coordinates of a point</td>
</tr>
<tr>
<td>Real</td>
<td>Floating point number</td>
</tr>
<tr>
<td>String</td>
<td>Sequence of characters</td>
</tr>
<tr>
<td>StringUtf8</td>
<td>String encoded in the UTF-8 format</td>
</tr>
<tr>
<td>Time</td>
<td>Time since midnight (in milliseconds)</td>
</tr>
<tr>
<td>TimeStamp</td>
<td>Date and time that includes combined date and time values</td>
</tr>
<tr>
<td>TimeStampInterval</td>
<td>Represents the difference between two timestamp values</td>
</tr>
</tbody>
</table>

Primitive types have a defined null value, which can be tested for by using the **null** language identifier; for example:

```
if d = null then
```

You can associate methods with primitive types, but you cannot associate properties with primitive types.

Properties that are defined as primitive types represent a value. They do not represent a reference to an object.

With the exception of the **Any** primitive type, which can represent any object reference or primitive value, the value of the property is stored in the parent object record when you define a property that is a primitive type. (A property that is an object contains a reference to the object.)

**Notes** A temporary value is created if the return value of a primitive type method is passed to an updating primitive method. On completion of the updating method, this temporary value is discarded.

You cannot specify the **clientExecution** and **serverExecution** method options on primitive type methods. Methods defined on primitive types are always executed in the node of the calling method.
Any Type

A variable of type **Any** can contain an object reference or any primitive value.

**Note**  The **Any** primitive type can be used only for local variables, parameters, and return types. You cannot define a property of type **Any**.

To determine the type of the value associated with a variable of type **Any**

- Use the **isKindOf** method.

The **Any** primitive type is useful when a:

- Method can return either an object reference or primitive value
- Parameter in a method can be either an object reference or a primitive value
- Variable can receive either an object reference or a primitive value

The following example shows the use of the **Any** primitive type.

```pascal
userNotification(eventType: Integer; userInfo: Any);
vars
    file : DbFile;
begin
    if eventType = 19 then
        file := userInfo.DbFile;
    ... 
    endif;
end;
```

For details about the methods defined in the **Any** primitive type, see "**Any Methods**", in the following subsection.

For details about converting primitive types, see "**Converting Primitive Types**", in Chapter 1 of the JADE Developer’s Reference.

### Any Methods

The methods defined in the **Any** primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns …</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>asString</strong></td>
<td>A string representing the value of a primitive type or the object id of an object reference</td>
</tr>
<tr>
<td><strong>display</strong></td>
<td>The string &quot;<strong>Any</strong>&quot;</td>
</tr>
<tr>
<td><strong>getName</strong></td>
<td>A string representing the value of a primitive type or the name of the receiver class if it is an object reference</td>
</tr>
<tr>
<td><strong>isKindOf</strong></td>
<td><strong>true</strong> if the type of the receiver is of the type specified by the <strong>type</strong> parameter</td>
</tr>
</tbody>
</table>
asString

Signature  asString(): String;

The `asString` method of the `Any` primitive type returns:

- A string representing the value of the primitive type, if the receiver is a primitive type
- A string containing the object identifier (oid) of the object reference, if the receiver is an object reference
- A null string, if the receiver is null

display

Signature  display(): String;

The `display` method of the `Any` primitive type returns the string "Any".

getName

Signature  getName(): String;

The `getName` method of the `Any` primitive type returns:

- A string containing the class name of the object reference, if the receiver is an object reference
- A string representing the value of a primitive type, if the receiver is a primitive type
- A null string, if the receiver is null

isKindOf

Signature  isKindOf(type: Type): Boolean;

The `isKindOf` method of the `Any` primitive type returns the `Boolean` value of `true` if the type of the `Any` variable is of the type specified by the `type` parameter. If the variable is a different type to that specified by the `type` parameter, the `isKindOf` method returns `false`.

The code fragment in the following example shows the use of the `isKindOf` method.

```plaintext
if not any.isKindOf(Object) then
    if any.isKindOf(Any) then
        return "not a valid reference";
    else
        return any.String;
    endif;
endif;
```

For example, `any.isKindOf(Integer)` returns `true` if the any variable contains an `Integer`, and `any.isKindOf(Customer)` returns `true` if the any variable contains a reference to an instance of the `Customer` class or one of its subclasses.

The following example shows the use of the `isKindOf` method.

```plaintext
vars
    date : Date;
begin
    date := fault.openDate;
```
if fault.isKindOf(GenuineFault) then
   opFault.value := true;
elseif ... then
   ...
endif;
end;
Binary Type

Use the **Binary** primitive type to define **Binary** variables and attributes.

When you specify a length less than or equal to **540** for a **Binary** attribute, it is embedded. Space is allocated within instances of the class to store a binary value with a length less than or equal to the specified length.

When you specify a length greater than **540** or you select the **Maximum Length** check box (which corresponds to **2,147,483,647** bytes) for a **Binary** attribute, it is not embedded. It is stored in a separate variable-length object, a Binary Large Object (blob), which can store a binary value with a length less than or equal to the specified length. The amount of storage required for a blob is determined by the binary value.

**Binary** variables can be bounded or unbounded, as shown in the following code fragment.

```
vars
    bin1 : Binary[100]; // Bounded - bin1 can store a binary value with a
                  // length less than or equal to 100 bytes
    bin2 : Binary;    // Unbounded - bin2 can store a binary value with a
                  // length less than or equal to 2,147,483,647 bytes
```

To specify a substring **bin[m:n]** of a **Binary** value **bin**, use two integers separated by a colon (:) character. The first integer is the start position and the second integer (following the colon (:) character) is the length of the binary substring or **end**, to indicate the end of the binary string. The first byte is at position 1.

A variable of type **Byte** can be used to reference a single byte in a binary value, in effect treating the binary value as an array of bytes, as shown in the following code fragment.

```
vars
    bin : Binary;
    byte : Byte;
begin
    bin := app.loadPicture("C:/Jade/bin/jade.bmp");
    byte := bin[716]; // 716th byte of the binary data in bin
```

For details about the constants and methods defined in the **Binary** primitive type, see "**Binary Constants**" and "**Binary Methods**", in the following subsections. For details about converting primitive types, see "**Converting Primitive Types**", in Chapter 1 of the **JADE Developer’s Reference**.

Binary Constants

The **Binary** primitive type provides the constants listed in the following table, for use with the **compressToBinary** methods in the **Binary**, **String**, and **StringUtf8** primitive types.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression_ZLib</td>
<td>1402</td>
<td>String and binary compression to binary using ZLIB level 5 (256*5 + 122)</td>
</tr>
<tr>
<td>Compression_ZLibFast</td>
<td>378</td>
<td>String and binary compression to binary using ZLIB level 1 (256*1 + 122)</td>
</tr>
<tr>
<td>Compression_ZLibSmall</td>
<td>2426</td>
<td>String and binary compression to binary using ZLIB level 9 (256*9 + 122)</td>
</tr>
</tbody>
</table>
## Binary Methods

The methods defined in the **Binary** primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns …</th>
</tr>
</thead>
<tbody>
<tr>
<td>ansiToString</td>
<td>The string equivalent of the binary interpreted as ANSI characters</td>
</tr>
<tr>
<td>ansiToUnicode</td>
<td>The Unicode string equivalent of the binary interpreted as ANSI characters</td>
</tr>
<tr>
<td>asDecimal</td>
<td>The <strong>Decimal</strong> representation of the receiver</td>
</tr>
<tr>
<td>asGuidString</td>
<td>A visual representation of the Globally Unique Identifier (GUID) binary receiver as a string of printable characters</td>
</tr>
<tr>
<td>base64Encode</td>
<td>An ASCII string consisting of lines with fewer than 76 characters resulting from encoding the receiver in Base64</td>
</tr>
<tr>
<td>base64EncodeNoCrLf</td>
<td>An ASCII string resulting from encoding the receiver in Base64 without carriage-return and line-feed characters</td>
</tr>
<tr>
<td>bufferAddress</td>
<td>The value of the pointer to the internal buffer as an integer</td>
</tr>
<tr>
<td>bufferMemoryAddress</td>
<td>The value of the pointer to the internal buffer as a memory address</td>
</tr>
<tr>
<td>compressToBinary</td>
<td>A compressed binary representation of the receiver</td>
</tr>
<tr>
<td>convertPicture</td>
<td>A copy of the receiver converted to the requested picture type</td>
</tr>
<tr>
<td>convertToFile</td>
<td>A copy of the receiver converted to the requested picture type in a file</td>
</tr>
<tr>
<td>display</td>
<td>A string containing a hexadecimal dump of the receiver</td>
</tr>
<tr>
<td>fromANSIToString</td>
<td>A String containing the receiver converted using the code page for the specified locale</td>
</tr>
<tr>
<td>fromANSIToStringUtf8</td>
<td>A UTF8 String containing the receiver converted using the code page for the specified locale</td>
</tr>
<tr>
<td>length</td>
<td>The actual length of a binary variable</td>
</tr>
<tr>
<td>maxLength</td>
<td>The declared maximum length of a binary variable</td>
</tr>
<tr>
<td>pictureSize</td>
<td>The type of image contained in the binary and the width and height of the image</td>
</tr>
<tr>
<td>pictureType</td>
<td>The type of picture image</td>
</tr>
<tr>
<td>posBinary</td>
<td>The position of a specified binary string in the receiver</td>
</tr>
<tr>
<td>posByte</td>
<td>The position of a specified byte in the receiver</td>
</tr>
<tr>
<td>unpackCString</td>
<td>A string extracted from the binary, starting at a specified position within the binary and terminated by the next occurring null character</td>
</tr>
<tr>
<td>uuidAsString</td>
<td>A string formatted as a Universally Unique Identifier (UUID)</td>
</tr>
</tbody>
</table>
**ansiToString**

**Signature**  
ansiToString(): String;

The `ansiToString()` method of the `Binary` primitive type interprets the binary as ANSI characters and returns a copy converted to a string.

When invoked from an ANSI application, an ANSI string is returned. When invoked from a Unicode application, the binary is converted from ANSI to Unicode, and a Unicode string is returned.

The code fragment in the following example shows the use of the `ansiToString()` method.

```pascal
str := bin.ansiToString;
```

**Note**  
When converting from ANSI to Unicode, conversion stops at the first null character. If the binary contains embedded nulls, the string returned from the ANSI to Unicode conversion therefore represents only that part of the binary up to the first null character.

**ansiToUnicode**

**Signature**  
ansiToUnicode(): String;

The `ansiToUnicode()` method of the `Binary` primitive type interprets the binary as ANSI characters and returns a copy converted to a Unicode string. This method can be invoked only from a Unicode application.

If this method is invoked from an ANSI application, the following exception is raised.

```pascal
1000 Invalid parameter type
```

**Note**  
Conversion of the binary stops at the first null character. If the binary contains embedded nulls, the string returned from the conversion therefore represents only that part of the binary up to the first null character.

**asDecimal**

**Signature**  
asDecimal(): Decimal;

The `asDecimal()` method of the `Binary` primitive type returns the decimal value for a `Binary` value that was obtained by a call to the `asBinary` method of the `Decimal` primitive type.

The following example shows the use of the `asDecimal` method.

```pascal
vars
    bin : Binary;
    dec : Decimal;
begin
    dec := 123.456.Decimal;
    bin := dec.asBinary;
    write bin.asDecimal; // Outputs 123.456
end;
```

Use the `asDecimal` method in preference to type casting; for example:

```pascal
    bin := dec.asBinary; // This is preferable to "bin := dec.Binary;"
asGuidString

**Signature** asGuidString(): String;

The `asGuidString` method of the Binary primitive type returns a visual representation of the Globally Unique Identifier (GUID) binary receiver as a string of printable characters, in the following format.

```
{{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}}
```

Binary class identifiers are used in ActiveX control and automation libraries, for example, and they take less space than a visual string representation. This method raises an exception if the receiver is not a valid GUID; that is, it is not a binary of length 16.

See also the String primitive type `asGuid` method.

base64Encode

**Signature** base64Encode(): String;

The `base64Encode` method of the Binary primitive type returns an ASCII string resulting from the encoding of the receiver using the Base64 encoding technique defined in RFC 1521.

Base64 encoding enables 8-bit data to be converted, so that it can be transmitted over a protocol that supports 7-bit characters only. Base64 encoding also provides enhanced privacy when the source data is standard ASCII text, as the message is no longer in clear text when it is transmitted.

The output string is represented in lines not exceeding 76 characters that are terminated with carriage return and line feed (Cr and Lf) characters.

Use the `base64Decode` method on the String primitive type to decode a Base64-encoded string.

The following example shows the use of the `base64Encode` method.

```plaintext
vars
  bin: Binary;
  file: File;
begin
  create file;
  file.fileName := "d:\temp\harry.jpg";
  file.kind := File.Kind_Binary;
  file.open;
  bin := file.readBinary(file.fileLength);
  write 'original length = ' & bin.length.String;
  write 'base64Encode length = ' & bin.base64Encode().length.String;
  write 'base64EncodeNoCrLf length = ' &
     bin.base64EncodeNoCrLf().length.String;
  write 'base64Decode length = ' &
     bin.base64Encode().base64Decode().length.String;
  write 'base64Decode length (from NoCrLf) = ' &
     bin.base64EncodeNoCrLf().base64Decode.length.String;
  file.close;
epilog
  delete file;
end;
```
Note  The length of an encoded string is about a third longer, even if the string is encoded with carriage-return and line-feed (Cr and Lf) characters.

**base64EncodeNoCrLf**

**Signature**  base64EncodeNoCrLf(): String;

The **base64EncodeNoCrLf** method of the **Binary** primitive type returns an ASCII string resulting from the encoding of the receiver using the Base64 encoding technique defined in RFC 1521.

Base64 encoding enables 8-bit data to be converted, so that it can be transmitted over a protocol that supports 7-bit characters only. Base64 encoding also provides enhanced privacy when the source data is standard ASCII text, as the message is no longer in clear text when it is transmitted.

Unlike the **base64Encode** method, the output is not broken up into lines; that is, it does not contain carriage-return and line-feed (Cr and Lf) characters.

Use the **base64Decode** method on the **String** primitive type to decode a Base64-encoded string.

The following example shows the use of the **base64EncodeCrLf** method.

```java
vars
  bin: Binary;
  file: File;
begin
  create file;
  file.fileName := "d:\temp\harry.jpg";
  file.kind := File.Kind_Binary;
  file.open;
  bin := file.readBinary(file.fileLength);
  write 'original length = ' & bin.length.String;
  write 'base64Encode length = ' & bin.base64Encode().length.String;
  write 'base64EncodeNoCrLf length = ' &
    bin.base64EncodeNoCrLf().length.String;
  write 'base64Decode length = ' &
    bin.base64Encode().base64Decode().length.String;
  write 'base64Decode length (from NoCrLf) = ' &
    bin.base64EncodeNoCrLf().base64Decode.length.String;
  file.close;
epilog
  delete file;
end;
```

**Note**  The length of an encoded string is about a third longer, even if the string is encoded with carriage-return and line-feed (Cr and Lf) characters.

**bufferAddress**

**Signature**  bufferAddress(): Integer;

The **bufferAddress** method of the **Binary** primitive type returns an integer containing the value of the pointer to the internal buffer that contains the binary. This value may be required when a JADE Binary primitive type value is being mapped to a structured record type for a call to an external function.
Call the \texttt{bufferAddress} method to determine the address of the buffer when an external function requires a data structure to contain a pointer to a second structure.

\textbf{Caution}  Do not use this method to pass the address of a binary to an external function that will be executed by a presentation client. If an external function is called from an application server method and executed by a different process (the presentation client), the memory address is not valid and will almost certainly result in a \texttt{jade.exe} (thin client) fault in the called function.

The method in the following example shows the use of the \texttt{bufferAddress} method to initialize the Windows \texttt{SECURITY_DESCRIPTOR} and \texttt{SECURITY_ATTRIBUTES} structures.

\begin{verbatim}
constants
  // Current security descriptor revision value
  SECURITY_DESCRIPTOR_REVISION = 1;
vars
  result : Boolean;
  securityDescriptor : Binary[20];
  securityAttributes : Binary[9];
begin
  ... // Call the Windows API to initialize the security descriptor
  result := call initializeSecurityDescriptor(securityDescriptor,
                                           SECURITY_DESCRIPTOR_REVISION);
  // Return Windows error if unable to initialize security descriptor
  if not result then
    return call getLastError;
  endif;
  // The first field (DWORD) in the security attributes structure is the
  // size (in bytes) of the structure
  // The second field (LPVOID) points to the security descriptor
  // Set the value to the actual address of the buffer
  ...
\end{verbatim}

\textbf{bufferMemoryAddress}

\textbf{Signature}  \texttt{bufferMemoryAddress}(): \texttt{MemoryAddress};

The \texttt{bufferMemoryAddress} method of the \texttt{Binary} primitive type returns a memory address containing the value of the pointer to the internal buffer that contains the binary. This value may be required when a JADE \texttt{Binary} primitive type value is being mapped to a structured record type for a call to an external function.

Call the \texttt{bufferMemoryAddress} method to determine the address of the buffer when an external function requires a data structure to contain a pointer to a second structure.

\textbf{Caution}  Do not use this method to pass the address of a binary to an external function that will be executed by a presentation client. If an external function is called from an application server method and executed by a different process (the presentation client), the memory address is not valid and will almost certainly result in a \texttt{jade.exe} (thin client) fault in the called function.
The method in the following example shows the use of the `bufferMemoryAddress` method to initialize the Windows `SECURITY_DESCRIPTOR` and `SECURITY_ATTRIBUTES` structures.

```plaintext
constants
    // Current security descriptor revision value
    SECURITY_DESCRIPTOR_REVISION = 1;
vars
    result    : Boolean;
    securityDescriptor : Binary[20];
    securityAttributes : Binary[9];
begin
    ... // Call the Windows API to initialize the security descriptor
    result := call initializeSecurityDescriptor(securityDescriptor,
                                                SECURITY_DESCRIPTOR_REVISION);
    // Return Windows error if unable to initialize security descriptor
    if not result then
        return call GetLastError;
    endif;
    // The first field (DWORD) in the security attributes structure is the
    // size (in bytes) of the structure
    // The second field (LPVOID) points to the security descriptor
    // Set the value to the actual address of the buffer
    securityAttributes[5:4] :=
        securityDescriptor.bufferMemoryAddress.asBinary32;
end;
```

### `compressToBinary`

**Signature**

```plaintext
compressToBinary(typeAndOption: Integer): Binary;
```

The `compressToBinary` method of the `Binary` primitive type returns a compressed binary representation of the binary of the receiver using the ZLIB compression value specified by the `typeAndOption` parameter, which can be one of the `Binary` primitive type constants listed in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression_ZLib</td>
<td>1402</td>
<td>String and binary compression to binary using ZLIB level 5 (256*5 + 122)</td>
</tr>
<tr>
<td>Compression_ZLibFast</td>
<td>378</td>
<td>String and binary compression to binary using ZLIB level 1 (256*1 + 122)</td>
</tr>
<tr>
<td>Compression_ZLibSmall</td>
<td>2426</td>
<td>String and binary compression to binary using ZLIB level 9 (256*9 + 122)</td>
</tr>
</tbody>
</table>

**Note** This method adds the type byte to the front of the compressed binary. This type byte is ignored when the value is used in a JADE system but if the data is to be passed to an external library, it is your responsibility to remove the type byte, if necessary.

You cannot concatenate the results of multiple `compressToBinary` method calls.

You must use the `Binary` primitive type `uncompressToBinary` method to uncompress a binary value from this binary representation.
**convertPicture**

**Signature**

```groovy
class BinaryType {
    method convertPicture(type: Integer): Binary;
}
```

The `convertPicture` method of the `Binary` primitive type returns a copy of a binary picture image converted to the picture type specified in the `type` parameter. The types of images that can be converted (by using the `Window` class `PictureType_Bitmap`, `PictureType_Jpeg`, `PictureType_Jpeg2000`, `PictureType_Png`, or `PictureType_Tiff` constant) are as follows.

- Bitmap (.bmp)
- Tag Image File Format (.tif)
- Joint Photographic Experts Group (.jpg)
- JPG 2000 (.jp2)
- Portable Network Graphics (.png)

A `14015` (File does not contain an image type that can be handled) exception is raised if the receiver does not contain valid image data.

**Notes**
Converts to a .tif image type file results in a tiff packbits type image. Converting to a .jpg image results in loss of quality in the picture, as Joint Photographic Experts Group (JPEG) uses a lossy compression algorithm.

As the Portable Network Graphics image uses a lossless compression algorithm, it provides clarity and retains definition for images, but the files may be larger than JPEG files.

An exception is raised if this method is invoked from a server method.

You cannot convert images to GIF picture files.

See also the `convertToFile` method of the `Binary` primitive type.

**convertToFile**

**Signature**

```groovy
class BinaryType {
    method convertToFile(filename: String; type: Integer);
}
```

The `convertToFile` method of the `Binary` primitive type saves a copy of a binary picture image converted to the picture type specified in the `type` parameter, in the file specified in the `filename` parameter. If the `filename` parameter is null (""), the common File Save dialog is invoked, requesting the file name that is to be used to store the converted image.

The types of images that can be converted are as follows.

- Bitmap (.bmp)
- Tag Image File Format (.tif)
- Joint Photographic Experts Group (.jpg)
- JPG 2000 (.jp2)
- Portable Network Graphics (.png)

Use the `PictureType_Bitmap`, `PictureType_Jpeg`, `PictureType_Jpeg2000`, `PictureType_Png`, or `PictureType_Tiff` constant of the `Window` class to specify the picture type.
A **14015** (File does not contain an image type that can be handled) exception is raised if the receiver does not contain valid image data.

**Notes** Converting to a .tiff image type results in a tiff packbits type image. Converting to a .jpg image results in loss of quality in the picture, as Joint Photographic Experts Group (JPEG) uses a *lossy* compression algorithm.

As the Portable Network Graphics image uses a *lossless* compression algorithm, it provides clarity and retains definition for images, but the files may be larger than JPEG files.

An exception is raised if this method is invoked from a server method.

You cannot convert images to GIF picture files.

See also the **convertPicture** method of the **Binary** primitive type.

**display**

**Signature**  
```plaintext
display(): String;
```

The **display** method of the **Binary** primitive type returns a string containing a hexadecimal dump of the receiver.

**fromANSIToString**

**Signature**  
```plaintext
fromANSIToString(lcid: Integer): String;
```

The **fromANSIToString** method of the **Binary** primitive type converts the receiver to a string using the code page for the locale specified by the **lcid** parameter and returns the resulting string.

Some code pages (for example, the one used in the People’s Republic of China locale) contain multi-byte characters as well as single-byte characters.

**fromANSIToStringUtf8**

**Signature**  
```plaintext
fromANSIToStringUtf8(lcid: Integer): StringUtf8;
```

The **fromANSIToStringUtf8** method of the **Binary** primitive type converts the receiver to a UTF8 string using the code page for the locale specified by the **lcid** parameter and returns the resulting string.

Some code pages (for example, the one used in the People’s Republic of China locale) contain multi-byte characters as well as single-byte characters.

**length**

**Signature**  
```plaintext
length(): Integer;
```

The **length** method of the **Binary** primitive type returns the actual length of the value that has been assigned to an embedded Binary property; for example, if you declared a **Binary** property with length of 30 but the value stored is of length 20, the **length** method returns 20.

**maxLength**

**Signature**  
```plaintext
maxLength(): Integer;
```

The **maxLength** method of the **Binary** primitive type returns the declared maximum length of a binary variable. If the binary variable maximum length has not been declared, the value of the **Max_UnboundedLength** global constant in the **SystemLimits** category is returned.
**pictureSize**

**Signature**

```
pictureSize(width: Integer output;
height: Integer output): Integer;
```

The `pictureSize` method of the `Binary` primitive type returns the type of picture image of the receiver and the width and height of the image. If the binary is not a valid image, zero (0) is returned for the type, width, and height of the image.

**Note** If the image contains multiple icon or cursor definitions, the `pictureSize` method returns the size of the largest of the images.

**pictureType**

**Signature**

```
pictureType(): Integer;
```

The `pictureType` method of the `Binary` primitive type returns the type of picture image. The return values are listed in the following table.

<table>
<thead>
<tr>
<th>Integer</th>
<th>Picture Type</th>
<th>Integer</th>
<th>Picture Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Not a valid picture</td>
<td>5</td>
<td>Cursor</td>
</tr>
<tr>
<td>1</td>
<td>Bitmap</td>
<td>6</td>
<td>Tag Image File Format (.tif)</td>
</tr>
<tr>
<td>2</td>
<td>Not used</td>
<td>7</td>
<td>Joint Photographic Experts Group (.jpg)</td>
</tr>
<tr>
<td>3</td>
<td>Icon</td>
<td>8</td>
<td>Portable Network Graphics (.png)</td>
</tr>
<tr>
<td>4</td>
<td>Metafile</td>
<td>9</td>
<td>Graphics Interchange Format (.gif)</td>
</tr>
</tbody>
</table>

You can use the `PictureType_Bitmap`, `PictureType_Icon`, `PictureType_MetaFile`, `PictureType_Cursor`, `PictureType_Tiff`, `PictureType_Jpeg`, `PictureType_Png`, or `PictureType_Gif` constant of the `Window` class, respectively, to specify the picture type.

An exception is raised if this method is invoked from a server method.

**posBinary**

**Signature**

```
pos(binary: Binary;
start: Integer): Integer;
```

The `posBinary` method of the `Binary` primitive type returns an integer containing the position in the receiver of the binary string specified in the `binary` parameter.

The `start` parameter must be greater than zero (0) and less than or equal to the length of the receiver.

**Note** This method is preferred to the deprecated `Binary` primitive type `pos` method.

**posByte**

**Signature**

```
posByte(b: Byte;
start: Integer): Integer;
```

The `posByte` method of the `Binary` primitive type returns an integer containing the position in the receiver of the byte specified in the `b` parameter.
The start parameter must be greater than zero (0) and less than or equal to the length of the receiver.

**Note** This method is preferred to the deprecated Binary primitive type pos method.

**uncompressToBinary**

**Signature**

```java
uncompressToBinary(): Binary;
```

The `uncompressToBinary` method of the Binary primitive type returns the uncompressed binary representation of the receiver.

This method uses the ZLIB compression routine specified in the `typeAndOption` parameter of the Binary primitive type `compressToBinary` method that was used to produce the compressed Binary value.

**uncompressToString**

**Signature**

```java
uncompressToString(): String;
```

The `uncompressToString` method of the Binary primitive type returns the uncompressed string representation of the receiver.

This method uses the ZLIB compression routine specified in the `typeAndOption` parameter of the String primitive type `compressToBinary` method that was used to produce the compressed Binary value.

**uncompressToStringUtf8**

**Signature**

```java
uncompressToStringUtf8(): StringUtf8;
```

The `uncompressToStringUtf8` method of the Binary primitive type returns the uncompressed UTF8 string representation of the receiver.

This method uses the ZLIB compression routine specified in the `typeAndOption` parameter of the StringUtf8 primitive type `compressToBinary` method that was used to produce the compressed Binary value.

**unicodeTo Ansi**

**Signature**

```java
unicodeToAnsi(): String;
```

The `unicodeToAnsi` method of the Binary primitive type interprets the binary as Unicode characters and returns a copy converted to an ANSI string.

This method can be invoked only from an ANSI application.

The code fragment in the following example shows the use of the `unicodeToAnsi` method.

```java
if fileIsUnicode then
    str := bin.unicodeToString;
else
    str := bin.ansiToString;
endif;
if str.length >= 3 and str[1:3] = "---" then
    rc := true;
endif;
```

If this method is invoked from a Unicode application, a 1068 - Feature not available in this release exception is raised.
Note  Conversion of the binary stops at the first null character. If the binary contains embedded nulls, the string returned from the conversion therefore represents only that part of the binary up to the first null character.

**unicodeToString**

*Signature*  unicodeToString(): String;

The `unicodeToString` method of the **Binary** primitive type interprets the binary as Unicode characters and returns a copy converted to a string.

The code fragment in the following example shows the use of the `unicodeToString` method.
```
if fileIsUnicode then
    binLen := bin[start : lenLen].unicodeToString.Integer;
else
    binLen := bin[start : lenLen].ansiToString.Integer;
endif;
```

When invoked from a Unicode application, a Unicode string is returned. When invoked from an ANSI application, the copy is converted from Unicode to ANSI, and an ANSI string is returned.

**Note**  When converting from Unicode to ANSI, conversion stops at the first null character. If the binary contains embedded nulls, the string returned from the Unicode to ANSI conversion therefore represents only that part of the binary up to the first null character.

**unpackCString**

*Signature*  unpackCString(start: Integer): String;

The `unpackCString` method of the **Binary** primitive type returns a string extracted from the binary, starting at the position specified by the `start` parameter and including all characters up to (but not including) the first null character.

The code fragment in the following example shows the use of the `unpackCString` method.
```
str := msg.unpackCString(1);
```

If the null character is not found, the string consists of all characters from the specified `start` position up to the end of the binary.

**Note**  Unpacking of the string stops at the first null character. If the C string contains embedded nulls, the returned string therefore represents only that part of the C string up to the first null character.

As the input is assumed to be a binary value of ANSI characters, the returned string is converted to Unicode characters when this method is used in a Unicode JADE system.

**uuidAsString**

*Signature*  uuidAsString(): String;

The `uuidAsString` method of the **Binary** primitive type returns a string formatted as a Universally Unique Identifier (UUID) from the receiver.
To be a valid UUID when calling this method, the binary should be 16 bytes. If it is less than 16 bytes, the value will be internally padded with zero bytes on the end, to make it 16 bytes long before the conversion is performed. If it is longer than 16 bytes, exception 1091 (Binary too long) is raised.

The `generateUuid` method of the `Application` class is used to generate a UUID, which has the Binary type.

The code fragment in the following example shows the use of the `uuidAsString` method.

```python
write self.uuid.uuidAsString;
```
Boolean Type

A Boolean primitive type value is one of the logical truth-values represented by the standard JADE identifiers true and false.

JADE provides standard operators that take Boolean values as operands, and produce a Boolean result. These operators include the logical:

- **and**
- **not** (negation)
- **or** (inclusive)

Boolean values can also be produced by applying relational operators to operands of other types. JADE provides the standard relational operators listed in the following table.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>

A relational operation consists of two operands separated by a relational operator. If the relation is satisfied, it has the value true. If the relation is not satisfied, it has the value false. The result of a relational operation is therefore a Boolean value.

The following example shows the use of the Boolean primitive type.

```plaintext
isMarried(): Boolean;
begin
    return spouse <> null;
end;
```

For details about the method defined in the Boolean primitive type, see "Boolean Method", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the JADE Developer’s Reference.

Boolean Method

The method defined in the Boolean primitive type is summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns …</th>
</tr>
</thead>
<tbody>
<tr>
<td>display</td>
<td>A string representing the value of the receiver</td>
</tr>
</tbody>
</table>
display

**Signature** display(): String;

The `display` method of the `Boolean` primitive type returns a string containing "true" or "false", depending on the value of the receiver.
Byte Type

A Byte primitive type value stores an unsigned integer value in the range 0 through 255.

JADE defines a number of arithmetic operations that take Integer operands and return Integer results, as listed in the following table. A Byte value can be used in place of an Integer value as an operand because of the implicit type conversion that takes place before the expression is evaluated. The result of an arithmetic operation involving Byte values is an Integer.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td>div</td>
<td>Integer division (division with truncation; for example, 7 div 3 = 2)</td>
</tr>
<tr>
<td>^</td>
<td>Exponentiation (for example, i ^ 3 is i cubed)</td>
</tr>
<tr>
<td>mod</td>
<td>Modulus (remainder after integer division)</td>
</tr>
</tbody>
</table>

These are binary (or dyadic) infix operators; that is, they are used with operands on both sides of the operator (for example, a+b). However, the + operator and - operator are also used as unary (or monadic) prefix operators, as listed in the following table.

<table>
<thead>
<tr>
<th>Unary Prefix Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+a</td>
<td>Sign identify</td>
</tr>
<tr>
<td>-a</td>
<td>Sign inversion</td>
</tr>
</tbody>
</table>

Using Byte Types in Assignments

An exception is raised when you attempt to compile an assignment of a numeric value (Decimal, Integer, Integer64, or Real) to a Byte variable without an explicit type cast, as shown in the following example.

```jade
vars
  byt : Byte;
  int : Integer;
begin
  byt := 123; // Not allowed to assign to a literal value
  int := 123; // Not allowed to assign to an Integer,
  byt := int; // or other numeric type
  byt := 123.Byte; // Allowed with the explicit type caste
  byt := int.Byte; // Allowed with the explicit type caste
end;
```

A runtime exception is raised if the value assigned to a Byte variable is less than 0 or greater than 255, as shown in the following example.

```jade
vars
  byt : Byte;
begin
```
A Byte value can be assigned to a variable of any of the numeric types (Integer, Integer64, Real, Decimal) without an explicit type cast.

For details about the methods defined in the Byte primitive type, see "Byte Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the JADE Developer’s Reference.

Byte Methods

The methods defined in the Byte primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitAnd</td>
<td>Returns a Byte value representing the receiver bits ANDed with the argument</td>
</tr>
<tr>
<td>bitNot</td>
<td>Returns a Byte value whose bit values are the inverse of the bit values of the receiver</td>
</tr>
<tr>
<td>bitOr</td>
<td>Returns a Byte value representing the receiver bits ORed with the argument</td>
</tr>
<tr>
<td>bitXor</td>
<td>Returns a Byte value representing the receiver bits XORed with the argument</td>
</tr>
<tr>
<td>display</td>
<td>Returns a string representing the value of the receiver</td>
</tr>
<tr>
<td>isEven</td>
<td>Returns true if the receiver represents an even number; otherwise false</td>
</tr>
<tr>
<td>isOdd</td>
<td>Returns true if the receiver represents an odd number; otherwise false</td>
</tr>
<tr>
<td>max</td>
<td>Returns the larger value of the receiver and a specified Byte</td>
</tr>
<tr>
<td>min</td>
<td>Returns the lesser value of the receiver and a specified Byte</td>
</tr>
<tr>
<td>numberFormat</td>
<td>Returns a string in the number format of the current locale</td>
</tr>
<tr>
<td>padLeadingWith</td>
<td>Returns the receiver as a string padded to the specified length with a leading character</td>
</tr>
<tr>
<td>parseCurrencyWithCurrentLocale</td>
<td>Sets the receiver to the result of parsing a string representing a currency value for the current locale</td>
</tr>
<tr>
<td>parseCurrencyWithFmtAndLcid</td>
<td>Sets the receiver to the result of parsing a string representing a currency value for the specified format and the specified locale</td>
</tr>
<tr>
<td>parseNumberWithCurrentLocale</td>
<td>Sets the receiver to the result of parsing a string representing a number for the current locale</td>
</tr>
<tr>
<td>parseNumberWithFmtAndLcid</td>
<td>Sets the receiver to the result of parsing a string representing a number for the specified format and the specified locale</td>
</tr>
<tr>
<td>userCurrencyFormat</td>
<td>Returns the receiver as a string in the specified currency format for the current locale</td>
</tr>
<tr>
<td>userCurrencyFormatAndLcid</td>
<td>Returns the receiver as a string in the specified currency format for the specified locale</td>
</tr>
<tr>
<td>userNumberFormat</td>
<td>Returns the receiver as a string in the specified number format for the current locale</td>
</tr>
</tbody>
</table>
### Encyclopaedia of Primitive Types

#### Byte Type

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>userNumberFormatAndLcid</code></td>
<td>Returns the receiver as a string in the specified number format for the specified locale</td>
</tr>
</tbody>
</table>

### bitAnd

**Signature**

```pascal
bitAnd(op: Byte): Byte;
```

The `bitAnd` method of the `Byte` primitive type compares each bit in the receiver with the corresponding bit specified in the `op` parameter and returns a `Byte` value representing the receiver bits ANDed with the argument.

The generated return values are listed in the following table.

<table>
<thead>
<tr>
<th>Bits in Receiver and Operand</th>
<th>Corresponding Bit in Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both bits are 1</td>
<td>1</td>
</tr>
<tr>
<td>One or both bits are not 1</td>
<td>0</td>
</tr>
</tbody>
</table>

The following example shows the use of the `bitAnd` method:

```pascal
keyDown(keyCode: Integer io; shift: Integer) updating;
  constants
  Shift = 1.Byte;
  Cntrl = 2.Byte;
  Alt = 4.Byte;
  vars
    byt : Byte;
  begin
    byt := shift.Byte;
    if byt.bitAnd(Shift) <> 0 then write "Shift key is down"; endif;
    if byt.bitAnd(Cntrl) <> 0 then write "Control key is down"; endif;
    if byt.bitAnd(Alt)    <> 0 then write "Alt key is down"; endif;
  end;
```

### bitNot

**Signature**

```pascal
bitNot(): Byte;
```

The `bitNot` method of the `Byte` primitive type returns a `Byte` value whose bit values are the inverse of the bit values of the receiver. The generated return values are listed in the following table.

<table>
<thead>
<tr>
<th>Bits in Receiver</th>
<th>Corresponding Bit in Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit is not 1</td>
<td>1</td>
</tr>
<tr>
<td>Bit is 1</td>
<td>0</td>
</tr>
</tbody>
</table>

### bitOr

**Signature**

```pascal
bitOr(op: Byte): Byte;
```

The `bitOr` method of the `Byte` primitive type compares each bit in the receiver with the corresponding bit specified in the `op` parameter, and returns a `Byte` value representing the receiver bits ORed with the argument.
The generated return values are listed in the following table.

<table>
<thead>
<tr>
<th>Bits in Receiver and Operand</th>
<th>Corresponding Bit in Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>One or both bits are 1</td>
<td>1</td>
</tr>
<tr>
<td>Neither bit is 1</td>
<td>0</td>
</tr>
</tbody>
</table>

The code fragment in the following example shows the use of the `bitOr` method.

```pascal
constants
    BitFlagNone = 0.Byte;
    BitFlag1   = 1.Byte;
vars
    byt : Byte;
begin
    byt := BitFlagNone;
    // set bit flag 1
    byt := byt.bitOr(BitFlag1);
    // test that bit flag 1 is set
    if byt.bitAnd(BitFlag1) <> 0 then
        write "flag 1 is set";
    endif;
end;
```

**bitXor**

Signature  `bitXor(op: Byte): Byte;`

The `bitXor` method of the `Byte` primitive type compares each bit in the receiver with the corresponding bit specified in the `op` parameter, and returns a `Byte` value representing the receiver bits XORed with the argument.

The generated return values are listed in the following table.

<table>
<thead>
<tr>
<th>Bits in Receiver and Operand</th>
<th>Corresponding Bit in Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>The bits are complementary</td>
<td>1</td>
</tr>
<tr>
<td>The bits are not complementary</td>
<td>0</td>
</tr>
</tbody>
</table>

**display**

Signature  `display(): String;`

The `display` method of the `Byte` primitive type returns a string that represents the integral value of the receiver.

**isEven**

Signature  `isEven(): Boolean;`

The `isEven` method of the `Byte` primitive type returns `true` if the receiver represents an even number; otherwise, it returns `false`. 
**isOdd**

**Signature**

\[
\text{isOdd(): Boolean;}
\]

The `isOdd` method of the `Byte` primitive type returns `true` if the receiver represents an odd number; otherwise, it returns `false`.

**max**

**Signature**

\[
\text{max(byte: Byte): Byte;}
\]

The `max` method of the `Byte` primitive type returns the larger value of the receiver and the value of the `byte` parameter. If the value of the receiver is greater than the value of the `byte` parameter, the value of the receiver is returned. If the value of the receiver is less than or equal to the value of the `byte` parameter, the value of `byte` is returned.

**min**

**Signature**

\[
\text{min(byte: Byte): Byte;}
\]

The `min` method of the `Byte` primitive type returns the lesser value of the receiver and the value of the `byte` parameter. If the value of the receiver is less than the value of the `byte` parameter, the value of the receiver is returned. If the value of the receiver is greater than or equal to the value of the `byte` parameter, the value of `byte` is returned.

**numberFormat**

**Signature**

\[
\text{numberFormat(): String;}
\]

The `numberFormat` method of the `Byte` primitive type returns a string in the numeric format defined for the current locale, which specifies the thousands separator, sign character, and decimal point character; for example, `129.00`. The following example shows the use of the `numberFormat` method.

```pascal
vars
  str : String;
  byt : Byte;
begin
  byt := 167.Byte;
  write byt;  // Outputs 167
  str := byt.numberFormat;  // Outputs 167.00
  write str;
end;
```

You can use the `defineNumberFormat` method of the `NumberFormat` class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
padLeadingWith

**Signature**

```
padLeadingWith(char: Character;
    max: Integer): String;
```

The `padLeadingWith` method of the `Byte` primitive type returns a string of the length specified in the `max` parameter, consisting of the receiving string padded with the leading character specified in the `char` parameter.

If the string is equal to or longer than the value specified in the `max` parameter, it is not truncated but the whole string is returned.

The following example shows the use of the `padLeadingWith` method.

```plaintext
constants
    PAD_CHARACTER = 'x';
vars
    byt : Byte;
    str : String;
begin
    byt := 123.Byte;
    str := byt.padLeadingWith('w', 15) & ' 678 Sesame St.';
    write str; // Outputs wwwwwwwwwwwwwww 678 Sesame St.
    str := byt.padLeadingWith('a', 2);
    write str; // Outputs 123
    str := byt.padLeadingWith(PAD_CHARACTER, 10);
    write str; // Outputs xxxxxxxx123
end;
```

parseCurrencyWithCurrentLocale

**Signature**

```
parseCurrencyWithCurrentLocale(source: String;
    errOffset: Integer output): Integer updating;
```

The `parseCurrencyWithCurrentLocale` method of the `Byte` primitive type parses the string specified in the `source` parameter to ensure that it matches the `Byte` format of the current locale for currency sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to zero (0).

This is equivalent to calling the `parseCurrencyWithFmtAndLcid` method, passing null in the `fmt` parameter and zero (0) in the `lcid` parameter.

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
parseCurrencyWithFmtAndLcid

Signature  parseCurrencyWithFmtAndLcid(source: String;
fmt: CurrencyFormat;
lcid: Integer;
errOffset: Integer output): Integer updating;

The parseCurrencyWithFmtAndLcid method of the Byte primitive type parses the string specified in the source parameter using the specified format and locale, to ensure that it matches the format specified in the fmt parameter for currency sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the fmt parameter is null, the settings for the locale specified in the lcid parameter are used. If the value of the lcid parameter is zero (0), the settings of the current locale are used.

If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output errOffset parameter, and sets the receiver to zero (0).

The currency character sequence is optional but if it is included in the source, it must be correctly positioned.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

A space included in the sign and currency character sequence is optional.

If native digits are allowed and the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The value of the source parameter text can include a decimal point and decimal digits, but they must all be zero (0) so that rounding or truncation is not required to store the value in a Byte variable; for example:

- "100", "100.", "100.0", "100.00", and "100.000" are accepted as valid and equal.
- "100.01" and "100.99" are rejected, as the value cannot be stored accurately in a Byte primitive type.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseNumberWithCurrentLocale

Signature  parseNumberWithCurrentLocale(source: String;
errOffset: Integer output): Integer updating;

The parseNumberWithCurrentLocale method of the Byte primitive parses the string specified in the source parameter to ensure that it matches the Byte format of the current locale for sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output errOffset parameter, and sets the receiver to zero (0).
This is equivalent to calling the `parseNumberWithFmtAndLcid` method, passing null in the `fmt` parameter and zero (0) in the `lcid` parameter.

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

### parseNumberWithFmtAndLcid

**Signature**

```
parseNumberWithFmtAndLcid(source: String;
fmt:   NumberFormat;
lcid:  Integer;
errOffset: Integer output): Integer updating;
```

The `parseNumberWithFmtAndLcid` method of the `Byte` primitive type parses the string specified in the `source` parameter using the specified format and locale, to ensure that it matches the format specified in the `fmt` parameter for sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the `fmt` parameter is null, the settings for the locale specified in the `lcid` parameter are used. If the value of the `lcid` parameter is zero (0), the settings of the current locale are used.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to zero (0).

The sign character sequence is optional but if it is included in the source, it must be correctly positioned. A space included in the sign sequence is optional.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The value of the `source` parameter text can include a decimal point and decimal digits, but they must all be zero so that rounding or truncation is not required to store the value in a `Byte` variable; for example:

- "100", "100.", "100.0", "100.00", and "100.000" are accepted as valid and equal.
- "100.01" and "100.99" are rejected, as the value cannot be stored accurately in a `Byte` primitive type.

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

### userCurrencyFormat

**Signature**

```
userCurrencyFormat(fmt: CurrencyFormat): String;
```

The `userCurrencyFormat` method of the `Byte` primitive type returns a string containing the receiver in the currency format specified in the `fmt` parameter.

To define your currency formats, use the Schema menu `Format` command from the Schema Browser.
Notes When you use a format in a JADE method, prefix your user currency format name with a dollar sign ($); for example, userCurrencyFormat($MyCurrency).

You can use the defineCurrencyFormat method of the CurrencyFormat class if you want to create your own transient format objects and define a currency format that dynamically overrides the format for the locale at runtime. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userCurrencyFormatAndLcid

Signature userCurrencyFormatAndLcid(fmt: CurrencyFormat; lcid: Integer): String;

The userCurrencyFormatAndLcid method of the Byte primitive type returns a string containing the receiver in the currency format and locale specified in the fmt parameter and lcid parameter, respectively.

If the value of the fmt parameter is null, the settings for the locale specified in the lcid parameter are used. If the value of the lcid parameter is zero (0), the settings of the current locale are used.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormat

Signature userNumberFormat(fmt: NumberFormat): String;

The userNumberFormat method of the Byte primitive type returns a string containing the receiver in the number format specified in the fmt parameter.

To define your numeric formats, use the Schema menu Format command from the Schema Browser.

Notes When you use a format in a JADE method, prefix your user number format name with a dollar sign ($); for example, userNumberFormat($MyNumber).

You can use the defineNumberFormat method from the NumberFormat class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at runtime. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
**userNumberFormatAndLcid**

**Signature**

```java
userNumberFormat(fmt: NumberFormat;
                 lcid: Integer): String;
```

The `userNumberFormatAndLcid` method of the `Byte` primitive type returns a string containing the receiver in the number format and locale specified in the `fmt` parameter and `lcid` parameter, respectively.

If the value of the `fmt` parameter is null, the settings for the locale specified in the `lcid` parameter are used. If the value of the `lcid` parameter is zero (0), the settings of the current locale are used.

If you do not define the `EnhancedLocaleSupport` parameter in the `JadeEnvironment` section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
Character Type

Use the Character primitive type to define a variable as a single ANSI or Unicode character. The following example shows the use of the Character primitive type.

```jade
testCharacter();
vars
  charValue : Character;
begin
  charValue := "M";       // Defines the variable value
  write charValue;        // Outputs a value of M
  write charValue.isLower; // Outputs a value of false
  write charValue.isAlpha; // Outputs a value of true
  write charValue.toLower; // Outputs a value of m
end;
```

For details about the methods defined in the Character primitive type, see "Character Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the JADE Developer's Reference.

Character Methods

The methods defined in the Character primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns …</th>
</tr>
</thead>
<tbody>
<tr>
<td>compareEq</td>
<td>true if the receiver is equal to a specified character</td>
</tr>
<tr>
<td>compareGeneric</td>
<td>An integer showing whether the receiver is greater than, equal to, or less than a specified character</td>
</tr>
<tr>
<td>compareGeq</td>
<td>true if the receiver is greater than or equal to a specified character</td>
</tr>
<tr>
<td>compareGrtr</td>
<td>true if the receiver is greater than a specified character</td>
</tr>
<tr>
<td>compareLeq</td>
<td>true if the receiver is less than or equal to a specified character</td>
</tr>
<tr>
<td>compareLss</td>
<td>true if the receiver is less than the value of a specified character</td>
</tr>
<tr>
<td>compareNeq</td>
<td>true if the receiver is not equal to a specified character</td>
</tr>
<tr>
<td>display</td>
<td>A string containing the receiver</td>
</tr>
<tr>
<td>isAlpha</td>
<td>true if the receiver represents a letter for the current language setting</td>
</tr>
<tr>
<td>isDelimiter</td>
<td>true if the receiver is not alphanumeric</td>
</tr>
<tr>
<td>isHex</td>
<td>true if the receiver represents a hexadecimal character</td>
</tr>
<tr>
<td>isLower</td>
<td>true if the receiver represents a lowercase letter</td>
</tr>
<tr>
<td>isNumeric</td>
<td>true if the receiver represents a numeric digit</td>
</tr>
<tr>
<td>isPrintable</td>
<td>true if the receiver is a character that can be printed</td>
</tr>
<tr>
<td>isUpper</td>
<td>true if the receiver represents an uppercase letter</td>
</tr>
<tr>
<td>makeString</td>
<td>A string of the specified length filled with the value of the receiver</td>
</tr>
<tr>
<td>setByteOrderLocal</td>
<td>A character that has the bytes ordered as required by the local node</td>
</tr>
</tbody>
</table>
Character Type

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns …</th>
</tr>
</thead>
<tbody>
<tr>
<td>setByteOrderRemote</td>
<td>A character that has the bytes ordered as required by the specified remote node</td>
</tr>
<tr>
<td>toHex</td>
<td>A string containing the hexadecimal value of the receiver</td>
</tr>
<tr>
<td>toLower</td>
<td>The lowercase equivalent of the receiving character</td>
</tr>
<tr>
<td>toUpper</td>
<td>The uppercase equivalent of the receiving character</td>
</tr>
</tbody>
</table>

**compareEql**

**Signature**

```plaintext
compareEql(rhs: Character;
          bIgnoreCase: Boolean;
          bUseLocale: Boolean;
          locale: Locale): Boolean;
```

The `compareEql` method of the `Character` primitive type returns `true` if the receiver is equal to the value of the `rhs` parameter; otherwise it returns `false`.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operator (=), documented in Chapter 1 of the JADE Developer’s Reference, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is `false`:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also `false`.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is `true`:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.
  ```plaintext
  recv.compareEql(lhs, true, false, null);
  recv.toLower = lhs.toLower;
  ```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.

- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the `compareEql` method.

```plaintext
write "A".compareEql("a", true, false, null);  // Outputs true
write "A".compareEql("a", false, false, null);  // Outputs false
```
**compareGeneric**

**Signature**
```
compareGeneric(rhs: Character;
    bIgnoreCase: Boolean;
    bUseLocale: Boolean;
    locale: Locale): Integer;
```

The `compareGeneric` method of the `Character` primitive type compares the receiver with the value of the `rhs` parameter and returns one of the following values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Returned if the receiver is …</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative integer</td>
<td>Less than the right-hand side value represented by the <code>rhs</code> parameter</td>
</tr>
<tr>
<td>Zero (0)</td>
<td>Equal to the right-hand side value represented by the <code>rhs</code> parameter</td>
</tr>
<tr>
<td>Positive integer</td>
<td>Greater than the right-hand side value represented by the <code>rhs</code> parameter</td>
</tr>
</tbody>
</table>

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operators `<, <=, =, >=, >, <>`, documented in Chapter 1 of the JADE Developer’s Reference, use a strict binary value comparison. If the value of the `bIgnoreCase` parameter is `false`:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also `false`.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is `true`:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.
  ```
  recv.compareGeneric(lhs, true, false, null);
  
  (recv.toLower>lhs.toLower).Integer - (recv.toLower<lhs.toLower).Integer;
  ```
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the `compareGeneric` method.

```java
vars
    locale : Locale;
begin
    write "a".compareGeneric("c", false, false, null);   // Outputs -1
    write "b".compareGeneric("b", false, false, null);   // Outputs 0
    write "c".compareGeneric("a", false, false, null);   // Outputs 1
```
// Comparisons with accented characters using binary and locale sort orders
locale := currentSchema.getLocale("5129");
write "à".compareGeneric("z", false, false, null); // Outputs 1
write "à".compareGeneric("z", false, true, locale); // Outputs -1

**compareGeq**

**Signature**

```plaintext
compareGeq(rhs: Character; bIgnoreCase: Boolean; bUseLocale: Boolean; locale: Locale): Boolean;
```

The **compareGeq** method of the **Character** primitive type returns `true` if the receiver is greater than or equal to the value of the `rhs` parameter; otherwise it returns `false`.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operator (>=), documented in Chapter 1 of the JADE Developer’s Reference, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is `false`:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also `false`.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is `true`:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```plaintext
recv.compareGeq(lhs, true, false, null);
recv.toLower >= lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the **compareGeq** method.

```plaintext
vars
    locale : Locale;
begin
    write "a".compareGeq("c", false, false, null); // Outputs false
    write "b".compareGeq("b", false, false, null); // Outputs true
    write "c".compareGeq("a", false, false, null); // Outputs true
    // Comparisons with accented characters using binary and locale sort orders
    locale := currentSchema.getLocale("5129");
```
The `compareGtr` method of the `Character` primitive type returns `true` if the receiver is greater than the value of the `rhs` parameter; otherwise, it returns `false`.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

The relational binary comparison operator (\(\gt\)), documented in Chapter 1 of the JADE Developer's Reference, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is `false`:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also `false`.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is `true`:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```java
recv.compareGtr(lhs, true, false, null);
recv.toLowerCase > lhs.toLowerCase;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the `compareGtr` method.

```java
vars
locale : Locale;
begin
write "a".compareGtr("z", false, false, null);    // Outputs false
write "a".compareGtr("z", false, true, locale);  // Outputs false
write "a".compareGtr("z", false, false, null);    // Outputs true
// Comparisons with accented characters using binary and locale sort orders
locale := currentSchema.getLocale("5129");
write "à".compareGtr("z", false, false, null);    // Outputs true
write "à".compareGtr("z", false, true, locale);  // Outputs false
```
**compareLeq**

**Signature**

```plaintext
compareLeq(lhs: Character;  
  bIgnoreCase: Boolean;  
  bUseLocale: Boolean;  
  locale: Locale): Boolean;
```

The `compareLeq` method of the `Character` primitive type returns `true` if the receiver is less than or equal to the value of the `rhs` parameter; otherwise, it returns `false`.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operator (`<=`), documented in Chapter 1 of the JADE Developer’s Reference, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is `false`:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also `false`.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is `true`:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```plaintext
recv.compareLeq(lhs, true, false, null);
recv.toLowerCase <= lhs.toLowerCase;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the `compareLeq` method.

```plaintext
vars
  locale : Locale;
begin
  write "a".compareLeq("c", false, false, null);  // Outputs true
  write "b".compareLeq("b", false, false, null);  // Outputs true
  write "c".compareLeq("a", false, false, null);  // Outputs false
  // Comparisons with accented characters using binary and locale sort orders
  locale := currentSchema.getLocale("5129");
  write "â".compareLeq("z", false, false, null);  // Outputs false
  write "â".compareLeq("z", false, true, locale);  // Outputs true
```
**compareLss**

**Signature**

```plaintext
compareLss(rhs: Character; bIgnoreCase: Boolean; bUseLocale: Boolean; locale: Locale): Boolean;
```

The **compareLss** method of the **Character** primitive type returns **true** if the receiver is less than the value of the `rhs` parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operator (<), documented in Chapter 1 of the JADE Developer’s Reference, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is **false**:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.
  ```plaintext
  recv.compareLss(lhs, true, false, null);
  recv.toLowerCase < lhs.toLowerCase;
  ```
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the `compareLss` method.

```plaintext
vars
locale : Locale;
begin
write "a".compareLss("c", false, false, null); // Outputs true
write "b".compareLss("b", false, false, null); // Outputs false
write "c".compareLss("a", false, false, null); // Outputs false
// Comparisons with accented characters using binary and locale sort orders
locale := currentSchema.getLocale("5129");
write "à".compareLss("z", false, false, null); // Outputs false
write "à".compareLss("z", false, true, locale); // Outputs true
```
**compareNeq**

*Signature*

```java
compareNeq(rhs: Character; bIgnoreCase: Boolean; bUseLocale: Boolean; locale: Locale): Boolean;
```

The `compareNeq` method of the `Character` primitive type returns `true` if the receiver is not equal to the value of the `rhs` parameter; otherwise, it returns `false`.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note**  The relational binary comparison operator (<>), documented in Chapter 1 of the JADE Developer’s Reference, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is `false`:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also `false`.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is `true`:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```
recv.compareNeq(lhs, true, false, null);
recv.toLower <> lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-insensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the `compareNeq` method.

```java
write "A".compareNeq("a", true, false, null); // Outputs false
write "A".compareNeq("a", false, false, null); // Outputs true
```

**display**

*Signature*

```java
display(): String;
```

If the receiver is a printable character, the `display` method of the `Character` primitive type returns a string containing the receiving character if it is a printable character for the current language setting.

If the receiver is not a printable character, the `display` method returns a string containing the hexadecimal value of the receiving character, enclosed in single quotation marks (") and preceded by a number sign (#).
The following example shows the use of the `display` method.

```pascal
vars
    char1, char2, char3 : Character;
begin
    char1 := "q";
    char2 := '#13';
    char3 := '#41';
    write char1.display; // Outputs q
    write char2.display; // Outputs #13'
    write char3.display; // Outputs A
end;
```

### isAlpha

**Signature**

```pascal
isAlpha(): Boolean;
```

The `isAlpha` method of the `Character` primitive type returns `true` if the receiver represents a letter for the current language setting; otherwise, it returns `false`.

The code fragment in the following example shows the use of the `isAlpha` method.

```pascal
//strip leading non-alpha characters
count := 1;
alphaFound := false;
while count <= str.length do
    if str[count].Character.isAlpha then
        if newStr.length > 0 then
            newStr := newStr & str[count].toUpper;
        else
            newStr := str[count].toUpper;
        endif;
        count := count + 1;
        alphaFound := true;
        break;
    endif;
    count := count + 1;
endwhile;
if alphaFound = false and newStr.length = 0 then
    return "";
end;
```

### isDelimiter

**Signature**

```pascal
isDelimiter(): Boolean;
```

The `isDelimiter` method of the `Character` primitive type returns `true` if the receiver is not alphanumeric; otherwise, it returns `false`.

The code fragment in the following example shows the use of the `isDelimiter` method.

```pascal
// find delimiter
count := int;
while count < self.length do
    charValue := self[count];
    if charValue.isDelimiter and charValue <> '_' then
```
Encyclopaedia of Primitive Types

Chapter 52

Character Type

EncycloPrim-7.1.08

break;
else
    count := count + 1;
endif;
endwhile;

isHex

Signature  isHex(): Boolean;

The isHex method of the Character primitive type returns true if the receiver represents a hexadecimal character; that is, 0 through 9, a through f, or A through F.

isLower

Signature  isLower(): Boolean;

The isLower method of the Character primitive type returns true if the receiver represents a lowercase letter for the current language setting.

The code fragment in the following example shows the use of the isLower method.

    if count <= str.length and str[count].Character.isLower then
        str[count] := str[count].Character.toUpper;
    endif;

isNumeric

Signature  isNumeric(): Boolean;

The isNumeric method of the Character primitive type returns true if the receiver represents a numeric digit for the current language setting.

The code fragment in the following example shows the use of the isNumeric method.

    if prodTitle.text.length = 0 then
        // Build default caption from name by inserting a space before each
        // uppercase character; for example, "3NeonHuedChocolateCoveredGumDrops"
        // becomes "3 Neon Hued Chocolate Covered Gum Drops"
        str := prodName.text;
        prodTitle.text[1] := str[1];
        count := 2;
        while count <= str.length do
            if str[count].isUpper or (str[count].isNumeric and not
            str[count-1].isNumeric) then
                prodTitle.text := prodTitle.text & " ";
            endif;
            prodTitle.text := prodTitle.text & str[i];
            count := count + 1;
        endwhile;
    endif;
isPrintable

**Signature**  
isPrintable(): Boolean;

The `isPrintable` method of the `Character` primitive type returns `true` if the receiver is a character that can be printed; otherwise, it returns `false`.

`Character` values in the range #20 through #7E can always be printed. Whether a character greater than #7F can be printed depends on the current locale, and character set (ANSI or Unicode).

isUpper

**Signature**  
isUpper(): Boolean;

The `isUpper` method of the `Character` primitive type returns `true` if the receiver represents an uppercase letter for the current language setting.

The code fragment in the following example shows the use of the `isUpper` method.

```java
// Check the first character is an uppercase letter
if not theName[1].Character.isUpper then
    app.beep;
    statusLine.caption := "Error - The name must begin with an uppercase letter";
    stringName.setFocus;
    return;
endif;
```

makeString

**Signature**  
makeString(length: Integer): String;

The `makeString` method of the `Character` primitive type returns a string of the length specified in the `length` parameter filled with the value of the receiver.

If the receiver is null, the returned string is filled with spaces. If the value of the `length` parameter is less than or equal to zero (0), an empty string is returned.

The following example shows the use of the `makeString` method.

```java
vars
    charValue : Character;
begin
    charValue := "*";
    write charValue.makeString(10);  // Outputs **********
    charValue := " ";
    write charValue.makeString(10);  // Outputs (ten spaces)
end;
```

setByteOrderLocal

**Signature**  
setByteOrderLocal(architecture: Integer): Character;

The `setByteOrderLocal` method of the `Character` primitive type returns a character that has the bytes ordered as required by the local node.
The bytes of the receiver are assumed to be ordered as indicated by the `architecture` parameter. The `architecture` parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the `getOSPlatform` method of the `Node` class.

The architecture can be one of the `Node` class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32BigEndian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32LittleEndian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64BigEndian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64LittleEndian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>

**Note** This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

**setByteOrderRemote**

**Signature**

```signature```
setByteOrderRemote(architecture: Integer): Character;
```signature```

The `setByteOrderRemote` method of the `Character` primitive type returns a character that has the bytes ordered as required by the remote node indicated by the `architecture` parameter.

The bytes of the receiver are assumed to be ordered as required by the local node.

The `architecture` parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the `getOSPlatform` method of the `Node` class.

The architecture can be one of the `Node` class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32BigEndian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32LittleEndian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64BigEndian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64LittleEndian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>

**Note** This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.
**toHex**

**Signature**    toHex(): String;

The `toHex` method of the `Character` primitive type returns a string containing the hexadecimal value of the receiving character.

The following example shows the use of the `toHex` method.

```pascal
vars
    charValue : Character;
begin
    charValue := "q";
    write charValue.toHex;   // Outputs 71
end;
```

**toLower**

**Signature**    toLower(): Character;

The `toLower` method of the `Character` primitive type returns the lowercase equivalent of the receiving character. Any character that is not an uppercase character is left unchanged.

The following example shows the use of the `toLower` method.

```pascal
vars
    charValue : Character;
begin
    charValue := "A";
    write charValue.toLower;   // Outputs a
end;
```

**toUpper**

**Signature**    toUpper(): Character;

The `toUpper` method of the `Character` primitive type returns the uppercase equivalent of the receiving character. Any character that is not a lowercase character is left unchanged.

The following example shows the use of the `toUpper` method.

```pascal
vars
    charValue : Character;
begin
    charValue := "r";
    write charValue.toUpper;   // Outputs R
end;
```
Date Type

A Date variable represents a specific day since the start of the Julian period. The Date primitive type defines the protocol for comparing and manipulating dates.

In JADE thin client mode, local variables of type Date are always initialized to the date relative to the presentation client.

Dates are generally obtained from user input in a Gregorian Calendar format, and converted to the internal (Julian day) format for internal storage and computation. Strictly speaking, the valid range for Julian day numbers is 0 through 2914726, which are the limits of the current Julian Period (7980 Julian years in length). This supports a valid date range from 24 November -4713 through 23 February 3268, Gregorian. However, JADE does not adhere to this limit, and allows day numbers to extend beyond this range.

JADE handles any calendar available within Microsoft Windows, including the conversion and display of non-Gregorian calendar dates, based on the locale and calendar set for the current user.

Notes  JADE prevents the use of the day 0 as a valid date, since the Julian day 0 is considered to be null. The maximum Julian day number is not imposed as the maximum date permitted by JADE.

write (1).Date // displays 25 November -4713 (one day higher than Day zero)
write (0).Date // displays null (but internally represents 24 November -4713)

The variable contains the current date as a Julian day number. (For details, see "Historical Note about the Date Type", in the following subsection.)

When used as a Dictionary key, the valid range for a Date key is (0).Date through (Max_Integer).Date.

If you declare a Date primitive type local variable in your method, it is initialized with the current date each time the method that declares the variable is invoked. Object attributes of type Date are initialized to null.

As Date primitive types are ordinal values (Julian day numbers), the forms of date arithmetic expressions listed in the following table are valid.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expression Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>date-expression + integer-expression</td>
<td>(date)</td>
</tr>
<tr>
<td>date-expression - integer-expression</td>
<td>(date)</td>
</tr>
<tr>
<td>date-expression - date-expression</td>
<td>(integer)</td>
</tr>
</tbody>
</table>

The following assignment shows the calculation of a date 134 days later than a specified date:

date2 := date1+134;

You can use the JadeEditMask class getTextAsDate and setTextFromDate methods to handle locale formatting for date fields.

For details about the methods defined in the Date primitive type, see "Date Methods", and for examples of the use of this primitive type, see "Date Primitive Type Examples", later in this section. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the JADE Developer's Reference.
Historical Note about the Date Type

The first official Gregorian calendar day occurred on the date of its inception on 15 October 1582 (Gregorian). This reform was later adopted by most western cultures at different times. We can still identify particular days before 15 October 1582 (Gregorian) using dates in the Gregorian calendar, simply by projecting the Gregorian dating system back beyond the time of its implementation. A calendar obtained by extension earlier in time than its invention or implementation is called the proleptic version of the calendar, and we therefore obtain the Proleptic Gregorian Calendar.

The Proleptic Gregorian Calendar has a year 0, and there are no years Before Christ (BC). The year before 1 Anno Domini (AD) is 0, and the year before that is -1.

The Julian day number of the Date primitive type is the number of elapsed days since the start of the Julian period, as defined by Joseph Scaliger. The start of the Julian period, established by Scaliger, is 1 January 4713 Before Common Era Proleptic Julian (BCE). The Julian period is the universal cycle (or period) used in chronology, especially for astronomical calculations involving large time intervals.

Date Primitive Type Examples

You can create a primitive method in type Date to return a string containing the short date in dd-MM-yyyy format, as follows.

```java
    testShortDate1(): String; begin
        return day.String & "-" & month.String & "-" & year.String;
    end;
```

In this example, the day, month, and year values are methods of the Date primitive type. However, the following example shows a more-direct method of returning the date in a short format; that is, the format method enables you to format dates to meet your requirements.

```java
    testShortDate2(): String; vars
        date : Date;
    begin
        return date.format("dd-MM-yyyy");
    end;
```

Notes  The month is denoted by uppercase letters (that is, MM) to differentiate it from minutes (that is, mm).

You can use the defineLongDateFormat or defineShortDateFormat method of the DateFormat class if you want to create your own transient format objects and define a long or short date format that dynamically overrides the appropriate format for the locale at run time. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

If an attempt is made to format an invalid date, "*invalid*" is returned.

The following example shows the use of the Date primitive type.

```java
    daysToXmas(): Integer;    // valid only before xmas day in a specified year vars
        xmasday, today : Date;
    begin
        xmasday.setDate(25, 12, today.year);
        return xmasday - today;
    end;
```
## Date Methods

The methods defined in the **Date** primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>day</code></td>
<td>Returns the day of the month as an integer</td>
</tr>
<tr>
<td><code>dayName</code></td>
<td>Returns the name of the day of the week as a string</td>
</tr>
<tr>
<td><code>dayNameWithLcid</code></td>
<td>Returns the name of the day of the week as a string for the specified locale</td>
</tr>
<tr>
<td><code>dayOfWeek</code></td>
<td>Returns an integer value for the day of the week</td>
</tr>
<tr>
<td><code>dayOfYear</code></td>
<td>Returns an integer value for the day of the year</td>
</tr>
<tr>
<td><code>display</code></td>
<td>Returns the date as a string</td>
</tr>
<tr>
<td><code>format</code></td>
<td>Returns the date as a string in the specified format</td>
</tr>
<tr>
<td><code>isFormattable</code></td>
<td>Returns <strong>true</strong> if the date falls within the valid conversion range for the execution platform</td>
</tr>
<tr>
<td><code>isLeapYear</code></td>
<td>Returns <strong>true</strong> if the year is a leap year</td>
</tr>
<tr>
<td><code>isValid</code></td>
<td>Returns <strong>true</strong> if the date is valid</td>
</tr>
<tr>
<td><code>longFormat</code></td>
<td>Returns the date as a string in the long date format</td>
</tr>
<tr>
<td><code>month</code></td>
<td>Returns the month as an integer</td>
</tr>
<tr>
<td><code>monthName</code></td>
<td>Returns the name of the month</td>
</tr>
<tr>
<td><code>monthNameWithLcid</code></td>
<td>Returns the name of the month for the specified locale</td>
</tr>
<tr>
<td><code>parseForCurrentLocale</code></td>
<td>Sets the receiver to the result of parsing a string representing a date for the current locale</td>
</tr>
<tr>
<td><code>parseLongWithCurrentLocale</code></td>
<td>Sets the receiver to the result of parsing a string representing a date in the long date format for the current locale</td>
</tr>
<tr>
<td><code>parseLongWithFmtAndLcid</code></td>
<td>Sets the receiver to the result of parsing a string representing a date for the specified long date format and the specified locale</td>
</tr>
<tr>
<td><code>parseLongWithPicAndLcid</code></td>
<td>Sets the receiver to the result of parsing a string representing a date for the specified long date picture and the specified locale</td>
</tr>
<tr>
<td><code>parseShortWithCurrentLocale</code></td>
<td>Sets the receiver to the result of parsing a string representing a date in the short date format for the current locale</td>
</tr>
<tr>
<td><code>parseShortWithFmtAndLcid</code></td>
<td>Sets the receiver to the result of parsing a string representing a date for the specified short date format and the specified locale</td>
</tr>
<tr>
<td><code>parseShortWithPicAndLcid</code></td>
<td>Sets the receiver to the result of parsing a string representing a date for the specified short date picture and the specified locale</td>
</tr>
<tr>
<td><code>setByteOrderLocal</code></td>
<td>Returns a date that has the bytes ordered as required by the local node</td>
</tr>
<tr>
<td><code>setByteOrderRemote</code></td>
<td>Returns a date that has the bytes ordered as required by the specified remote node</td>
</tr>
<tr>
<td><code>setDate</code></td>
<td>Sets the receiver to a specified date</td>
</tr>
<tr>
<td><code>setDateYearAbsolute</code></td>
<td>Sets the receiver to a specific absolute date</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>shortDayNameWithLcid</td>
<td>Returns the short name of the week day for the specified locale</td>
</tr>
<tr>
<td>shortFormat</td>
<td>Returns the date in the short date format</td>
</tr>
<tr>
<td>shortMonthNameWithLcid</td>
<td>Returns the short name of the month for the specified locale</td>
</tr>
<tr>
<td>subtract</td>
<td>Returns the interval between the receiver and the specified date</td>
</tr>
<tr>
<td>userFormat</td>
<td>Returns the date as a string in the specified date format</td>
</tr>
<tr>
<td>userLongFormatAndLcid</td>
<td>Returns the date as a string in the specified long date format for the specified locale</td>
</tr>
<tr>
<td>userLongFormatPicAndLcid</td>
<td>Returns the date as a string formatted in the specified long date picture for the specified locale</td>
</tr>
<tr>
<td>userShortFormatAndLcid</td>
<td>Returns the date as a string in the specified short date format for the specified locale</td>
</tr>
<tr>
<td>userShortFormatPicAndLcid</td>
<td>Returns the date as a string formatted in the specified short date picture for the specified locale</td>
</tr>
<tr>
<td>year</td>
<td>Returns the year as in integer</td>
</tr>
</tbody>
</table>

**day**

**Signature**

day(): Integer;

The `day` method of the `Date` primitive type returns the day of the month (represented by the date value of the receiver) as an integer; for example, 21.

The following code fragment shows the use of the `day` method.

```java
while day.day <= Calendar.DaysOfWeek do
    column := day.dayOfWeek;
    text := day.dayName[1:Calendar.DayNameLength];
    day := day + 1;
endwhile;
result := day.day.String & "/" & date.month.String & "/" & date.year.String[3:2];
```

**dayName**

**Signature**

dayName(): String;

The `dayName` method is defined by the current locale of the user. The `dayName` method of the `Date` primitive type returns the name of the day of the week (represented by the date value of the receiver) as a string; for example, Tuesday.

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

For an example of the use of the `dayName` method, see the `Date` primitive type `day` method.
dayNameWithLcid

**Signature**  
dayNameWithLcid(lcid: Integer): String;

The `dayNameWithLcid` method of the Date primitive type returns a string containing the full name of the week day from the locale specified in the `lcid` parameter for the receiver date.

If the value of the `lcid` parameter is zero (0), the day name is obtained from the current locale. If the date is null or invalid, an exception is raised.

If you do not define the `EnhancedLocaleSupport` parameter in the `[JadeEnvironment]` section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

dayOfWeek

**Signature**  
dayOfWeek(): Integer;

The `dayOfWeek` method of the Date primitive type returns an integer value for the day of the week (for the date value of the receiver).

For an example of the use of the `dayOfWeek` method, see the Date primitive type `day` method.

The week day values are those listed in the following table.

<table>
<thead>
<tr>
<th>Integer Value</th>
<th>Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Monday</td>
</tr>
<tr>
<td>2</td>
<td>Tuesday</td>
</tr>
<tr>
<td>3</td>
<td>Wednesday</td>
</tr>
<tr>
<td>4</td>
<td>Thursday</td>
</tr>
<tr>
<td>5</td>
<td>Friday</td>
</tr>
<tr>
<td>6</td>
<td>Saturday</td>
</tr>
<tr>
<td>7</td>
<td>Sunday</td>
</tr>
</tbody>
</table>

If you do not define the `EnhancedLocaleSupport` parameter in the `[JadeEnvironment]` section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

dayOfYear

**Signature**  
dayOfYear(): Integer;

The `dayOfYear` method of the Date primitive type returns an integer value for the day of the year (for the date value of the receiver); for example, 274.
The following example shows the use of the **dayOfYear** method.

```plaintext
vars
date : Date;
begin
date := "01 January 2000".Date;
write date.dayOfYear.String; // Outputs 1
date := "10 March 2000".Date;
write date.dayOfYear.String; // Outputs 70
end;
```

**display**

**Signature**

```plaintext
display(): String;
```

The **display** method of the **Date** primitive type returns the receiver as a string.

**format**

**Signature**

```plaintext
format(picture: String): String;
```

The **format** method of the **Date** primitive type returns the receiver as a string formatted in the date format specified in the **picture** parameter.

**Notes**

If the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is set to its default value of **false**, the **format** method returns "invalid" for dates outside the range 1 January 1601 through 31 December 30827.

If **EnhancedLocaleSupport** is set to **true**, the **format** method can format dates in the range 1 January 100 to 31 December 30827 correctly.

If **EnhancedLocaleSupport** is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The following examples show the use of the **format** method.

```plaintext
vars
date : Date;
begin
write "The date today is " & date.format("dd.MM.yyyy");
end;
```

```plaintext
if cd.lastPlayed = null then
form.lastPlayed.caption := "Never";
else
form.lastPlayed.caption := cd.lastPlayed.format("dd-MM-yy");
endif;
```

The **picture** parameter is the string value of the required format. The month is denoted by uppercase letters (that is, **MM**) to differentiate it from minutes (that is, **mm**).
Use the string picture elements listed in the following table to construct date format picture strings. Separate each element with a space or separator character.

<table>
<thead>
<tr>
<th>Picture</th>
<th>Description</th>
<th>Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Day of month as digits, with no leading zero</td>
<td>9</td>
</tr>
<tr>
<td>dd</td>
<td>Day of month as digits, with a leading zero</td>
<td>09</td>
</tr>
<tr>
<td>ddd</td>
<td>Day of week as an abbreviation as specified in the locale definition, usually three letters</td>
<td>Mon</td>
</tr>
<tr>
<td>ddddd</td>
<td>Day of week as the full name</td>
<td>Wednesday</td>
</tr>
<tr>
<td>M</td>
<td>Month as digits, with no leading zero</td>
<td>6</td>
</tr>
<tr>
<td>MM</td>
<td>Month as digits, with leading zero</td>
<td>08</td>
</tr>
<tr>
<td>MMM</td>
<td>Month as an abbreviation as specified in the locale definition, usually three letters</td>
<td>Mar</td>
</tr>
<tr>
<td>MMMM</td>
<td>Month as the full name</td>
<td>September</td>
</tr>
<tr>
<td>y</td>
<td>Year, represented by only the last digit if less than 10, else yy</td>
<td>6</td>
</tr>
<tr>
<td>yy</td>
<td>Year, represented by only the last two digits</td>
<td>97</td>
</tr>
<tr>
<td>yyy</td>
<td>Year, represented by all significant digits</td>
<td>1998</td>
</tr>
</tbody>
</table>

For example, to return a date string of **Wed Aug 04 99**, use the following picture string as your picture parameter for the format method of the Date primitive type.

```
ddd MMM dd yy
```

**Note**  
The locale information for each country is supplied embedded with the operating system. For example, the **MMMM** format of an English locale returns the month as a three-letter abbreviation (for example, **Apr**) and the **ddd** format as a three-letter abbreviation (for example, **Mon**). The **MMMM** format of a French locale can vary from three letters to four letters and a character (for example, **mai**, **mars**, or **janv.**) and the **ddd** format always returns three letters and a character (for example, **lun.**).

**isFormatable**

**Signature**  
```
isFormatable(): Boolean;
```

The **isFormatable** method of the **Date** primitive type returns **true** for dates that can be formatted correctly by using the **shortFormat** and **longFormat** methods of the **Date** primitive type; otherwise it returns **false**.

The formatting methods in JADE depend on the operating system date conversion routines, which result in the **isFormatable** method returning **true** for dates in the range **1 January 1601** through **31 December 30827**.

**isLeapYear**

**Signature**  
```
isLeapYear(): Boolean;
```

The **isLeapYear** method of the **Date** primitive type returns **true** if the year of the locale and calendar that is set for the current user is a leap year.
isValid

Signature isValid(): Boolean;

The isValid method of the Date primitive type returns true if the receiver is a valid date. This method returns false if the date is invalid.

This method also returns false for dates outside the valid internal representation range of 24th November -4713 through 31st December 1465072 Gregorian.

The code fragment in the following example shows the use of the isValid method.

```java
if (day.String & "/" & month.String & "/" & year.String).Date.isValid then
    calendar.date.setDate(day, month, year);
else
    calendar.date.setDate(1, month, year);
endif;
```

longFormat

Signature longFormat(): String;

The longFormat method of the Date primitive type is defined by the current locale of the user.

The longFormat method returns the receiver as a string formatted in the long date format defined for the current locale of the user; for example, Wednesday, 04 08, 99 or Wed, August 4, 1999.

Notes You can use the defineLongDateFormat method of the DateFormat class if you want to create your own transient format objects and define a long date format that dynamically overrides the format for the locale at runtime. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

The longFormat method returns "*invalid*" for dates before 1601 or after 30827.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

month

Signature month(): Integer;

The month method of the Date primitive type returns the month (represented by the date value of the receiver) as an integer; for example, 08.

The code fragment in the following example shows the use of the month method.

```java
if date <> null then
    if date.month <> value.month or date.year <> value.year then
        monthTable.showDays(value);
    endif;
endif;
```
**monthName**

**Signature**  
`monthName(): String;`

The `monthName` method of the `Date` primitive type returns the name of the month (for the date value of the receiver) as a string.

The following example shows the use of the `monthName` method.

```java
date(set: Boolean; value: Date io) mapping, updating;
begin
  if set and isTransient and date <> value then
    if date <> null then
      if date.month <> value.month or date.year <> value.year then
        monthTable.showDays(value);
      endif;
    endif;
    monthTable.selectDay(value);
    monthLabel.caption := value.monthName;
    yearLabel.caption := value.year.String;
    date := value;
    if changeType = ChangeType_None then
      changeType := ChangeType_Script;
    endif;
    click(self);
    changeType := ChangeType_None;
  endif;
end;
```

If you do not define the `EnhancedLocaleSupport` parameter in the `[JadeEnvironment]` section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**monthNameWithLcid**

**Signature**  
`monthNameWithLcid(lcid: Integer): String;`

The `monthNameWithLcid` method of the `Date` primitive type returns a string containing the full name of the month from the locale specified in the `lcid` parameter for the receiver date.

If the value of the `lcid` parameter is zero (0), the month name is obtained from the current locale. If the date is null or invalid, an exception is raised.

If you do not define the `EnhancedLocaleSupport` parameter in the `[JadeEnvironment]` section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
parseForCurrentLocale

Signature parseForCurrentLocale(inputDateString: String): Boolean updating;

The parseForCurrentLocale method of the Date primitive type assigns a date to the receiver based on the string specified in the inputDateString parameter. The string is assumed to be formatted using the conventions of your current locale. (Define the date convention for your locale by using the Date sheet of the Regional Settings Properties dialog, accessed from the Regional Settings icon in the Control Panel.)

This method returns true if the string represents a valid date; otherwise it returns false. This method handles strings in long or short date format, as shown in the following examples.

27 August 2011
Monday, 27 August 2011
2010-10-29
08/27/2010
27 Aug 99
20 noiembrie 1999 // the current locale is set to Romanian, for example

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. By default, formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

Non-English day and month names are handled; that is, locales that have short names with more than three characters or names with non-alphanumeric characters (for example, French and Thai).

parseLongWithCurrentLocale

Signature parseLongWithCurrentLocale(source: String;
            errOffset: Integer output): Integer updating;

The parseLongWithCurrentLocale method of the Date primitive type parses the string specified in the source parameter to ensure that it matches the long date format of the current locale in terms of element (day, month, year) order and separators and if valid, assigns the value to the receiver.

If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output errOffset parameter, and sets the receiver to the invalid date value (the isValid method of the Date primitive type will return false).

This is equivalent to calling the parseLongWithFmtAndLcid method, passing null in the fmt parameter and zero (0) in the lcid parameter.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
parseLongWithFmtAndLcid

Signature  parseLongWithFmtAndLcid(source: String; fmt: DateFormat; lcid: Integer; errOffset: Integer output): Integer updating;

The `parseLongWithFmtAndLcid` method of the `Date` primitive type parses the string specified in the `source` parameter to ensure that it matches the long date format specified in the `fmt` parameter in terms of element (day, month, year) order and separator.

Text such as day and month names must match the appropriate values for the locale specified in the `lcid` parameter. If the source string contains a valid date, it is assigned to the receiver; otherwise the invalid date value is assigned to the receiver (the `isValid` method of the `Date` primitive type will return `false`).

If the value of the `fmt` parameter is null, the long date format of the locale specified in the `lcid` parameter is used. If the value of the `lcid` parameter is zero (0), the long date format of the current locale is used.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to the invalid date value.

This method is the same as the `parseLongWithPicAndLcid` method except that the picture string is taken from the `DateFormat` class `longFormat` property. For more details and examples of valid date matches, see the `parseLongWithPicAndLcid` method.

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseLongWithPicAndLcid

Signature  parseLongWithPicAndLcid(source: String; pic: String; lcid: Integer; errOffset: Integer output): Integer updating;

The `parseLongWithPicAndLcid` method of the `Date` primitive type parses the string specified in the `source` parameter to ensure that it matches the long date picture string specified in the `pic` parameter in terms of element (day, month, year) order and separators.

Text such as day and month names must match the appropriate values for the locale specified in the `lcid` parameter. If the source string contains a valid date, it is assigned to the receiver; otherwise the invalid date value is assigned to the receiver (the `isValid` method of the `Date` primitive type will return `false`).

If the value of the `pic` parameter is null, the long date format picture of the locale specified in the `lcid` parameter is used. If the value of the `lcid` parameter is zero (0), the long date format picture of the current locale is used.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to the invalid date value.

The picture element is defined in the `picture` parameter of the `Date` primitive type `format` method.
The following table shows examples of valid matches between source text and format in the English (NZ) locale.

<table>
<thead>
<tr>
<th>Date Value (Specified by the source Parameter)</th>
<th>Format (Specified by the pic Parameter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thursday, 1/July/2010</td>
<td>dddd, d/MMM/yyyy</td>
</tr>
<tr>
<td>1/July/2010</td>
<td>d/MMM/yyyy</td>
</tr>
<tr>
<td>1/7/2010</td>
<td>d/M/yyyy</td>
</tr>
<tr>
<td>1 July 2010, Thursday</td>
<td>d MMMM yyyy, dddd</td>
</tr>
<tr>
<td>1/7/2010, Thu</td>
<td>d/M/yyyy, ddd</td>
</tr>
<tr>
<td>1/Jul/2001</td>
<td>d/MMM/yyyy</td>
</tr>
<tr>
<td>1 Jul 2001</td>
<td>d MMM yyyy</td>
</tr>
</tbody>
</table>

Day and month names are matched with the locale table entries using a locale-driven case-insensitive comparison. Abbreviated names are considered to be valid matches for full name specifiers.

**Note** The date is assumed to be localized Gregorian. Other calendars such as Hebrew Lunar are not supported.

If the first character of the source is a digit, the characters in the long date format picture up to but not including the first day number, month, or year specifier are considered optional. The source can include a month number, where a month name specifier occurs. Leading zeros for day and month numbers are optional when day number, month, and year have intervening separators.

Provided the source text includes values for each day number, month, and year and the source text ends with the last element, any trailing text in the format picture (for example, separators and day name) is considered optional.

If the year designator is "yy", exactly two digits must be present for the year number. The two-digit value is upgraded to a four-digit value using the `CAL_ITWODIGITYEARMAX` setting, which you can set and modify in the `Regional` settings of the Windows Control Panel. Its default value is **2029**, which gives the following conversions.

- 00…29 becomes 2000…2029
- 30…99 becomes 1930…1999

All other year designators require that exactly four digits must be specified for the year number and its value must be in the range zero (0) through 30,000. Negative values (implying the Gregorian BC era) are not supported. If the locale is Thai, the year is assumed to include the Thai year offset (that is, 543), which is subtracted from the date assigned to the receiver.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

All characters in the picture string other than the format specifiers (d, M, y, and g) must be matched exactly in the source string. For example, if the format picture string includes some characters before the first specifier, the source must include those exact characters before the first day digit.

**Note** Single quote characters (‘) can be used to enclose literal characters, including the format specifier characters. A pair of single quotes occurring in single quoted text is treated as one character. For example, to display the date as "May '93", the format string is "MMMM "’yy". The first and last single quotation marks are the enclosing quotation marks. The second and third single quotation marks are the escape sequence, to allow the single quotation mark to be displayed before the century.
If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseShortWithCurrentLocale**

```
Signature  parseShortWithCurrentLocale(source:  String;
        errOffset: Integer output): Integer updating;
```

The `parseShortWithCurrentLocale` method of the `Date` primitive type parses the string specified in the `source` parameter to ensure that it matches the short date format of the current locale in terms of element (day, month, year) order and separators and if valid, assigns the value to the receiver.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to the invalid date value (the `isValid` method of the `Date` primitive type will return false).

This is equivalent to calling the `parseShortWithFmtAndLcid` method, passing null in the `fmt` parameter and zero (0) in the `lcid` parameter.

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseShortWithFmtAndLcid**

```
Signature  parseShortWithFmtAndLcid(source:  String;
        fmt:  DateFormat;
        lcid:  Integer;
        errOffset: Integer output): Integer updating;
```

The `parseShortWithFmtAndLcid` method of the `Date` primitive type parses the string specified in the `source` parameter to ensure that it matches the short date format specified in the `fmt` parameter in terms of element (day, month, year) order and separators.

Text such as day and month names must match the appropriate values for the locale specified in the `lcid` parameter. If the source string contains a valid date, it is assigned to the receiver; otherwise the invalid date value is assigned to the receiver (the `isValid` method of the `Date` primitive type will return false).

If the value of the `fmt` parameter is null, the short date format of the locale specified in the `lcid` parameter is used. If the value of the `lcid` parameter is zero (0), the short date format of the current locale is used.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to the invalid date value.

This method is the same as the `parseShortWithPicAndLcid` method except that the picture string is taken from the `DateFormat` class `shortFormat` property. For more details and examples of valid date matches, see the `parseShortWithPicAndLcid` method.
If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseShortWithPicAndLcid**

**Signature**

```
parseShortWithPicAndLcid(source: String; pic: String; lcid: Integer; errOffset: Integer output): Integer updating;
```

The `parseShortWithPicAndLcid` method of the Date primitive type parses the string specified in the source parameter to ensure that it matches the short date picture string specified in the pic parameter in terms of element (day, month, year) order and separators.

Text such as day and month names must match the appropriate values for the locale specified in the lcid parameter. If the source string contains a valid date, it is assigned to the receiver; otherwise the invalid date value is assigned to the receiver (the isValid method of the Date primitive type will return false).

If the value of the pic parameter is null, the short date format picture of the locale specified in the lcid parameter is used. If the value of the lcid parameter is zero (0), the short date format picture of the current locale is used.

If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output errOffset parameter, and sets the receiver to the invalid date value. The picture element is defined in the picture parameter of the Date primitive type format method.

The following are examples of valid matches between source text and format in the English (NZ) locale.

<table>
<thead>
<tr>
<th>Date Value (Specified by the source Parameter)</th>
<th>Format (Specified by the pic Parameter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/07/2001</td>
<td>ddMMyyyy</td>
</tr>
<tr>
<td>1 Jul 2010</td>
<td>d MMM yyyy</td>
</tr>
<tr>
<td>1/Jul/2010</td>
<td>d/MMM/yyyy</td>
</tr>
<tr>
<td>1/7/2010</td>
<td>d/MM/yyyy</td>
</tr>
<tr>
<td>1/07/2001</td>
<td>d/MM/yyyy</td>
</tr>
</tbody>
</table>

Day and month names are matched with the locale table entries using a locale-driven case-insensitive comparison. Abbreviated names are considered to be valid matches for full name specifiers.

**Note** The date is assumed to be localized Gregorian. Other calendars such as Hebrew Lunar are not supported.

If the first character of the source is a digit, the characters in the short date format picture up to but not including the first day number, month, or year specifier are considered optional. The source can include a month number, where a month name specifier occurs. Leading zeros for day and month numbers are optional when day number, month, and year have intervening separators.

Provided the source text includes values for each of day number, month, and year and the source text ends with the last element, any trailing text in the format picture (for example, separators and day name) is considered optional.
If the year designator is "yy", exactly two digits must be present for the year number. The two-digit value is upgraded to a four-digit value using the CAL_ITWODIGITYEARMAX setting, which you can set and modify in the Regional settings of the Windows Control Panel. Its default value is 2029, which gives the following conversions.

- 00…29 becomes 2000…2029
- 30…99 becomes 1930…1999

All other year designators require that exactly four digits must be specified for the year number and its value must be in the range zero (0) through 30,000. Negative values (implying the Gregorian BC era) are not supported. If the locale is Thai, the year is assumed to include the Thai year offset (that is, 543), which is subtracted from the date assigned to the receiver.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

All characters in the picture string other than the format specifiers (d, M, y, and g) must be matched exactly in the source string. For example, if the format picture string includes some characters before the first specifier, the source must include those exact characters before the first day digit.

**Note** Single quote characters (‘) can be used to enclose literal characters, including the format specifier characters. A pair of single quotes occurring in single quoted text is treated as one character. For example, to display the date as "May '93", the format string is "MMM "'yy". The first and last single quotation marks are the enclosing quotation marks. The second and third single quotation marks are the escape sequence, to allow the single quotation mark to be displayed before the century.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**setByteOrderLocal**

**Signature**

```java
setByteOrderLocal(architecture: Integer): Date;
```

The `setByteOrderLocal` method of the `Date` primitive type returns a date that has the bytes ordered as required by the local node. The bytes of the receiver are assumed to be ordered as indicated by the `architecture` parameter.

The `architecture` parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the `getOSPlatform` method of the `Node` class.

The architecture can be one of the `Node` class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>
**setByteOrderRemote**

**Signature**

```java
setByteOrderRemote(architecture: Integer): Date;
```

The `setByteOrderRemote` method of the `Date` primitive type returns a date that has the bytes ordered as required by the remote node indicated by the `architecture` parameter.

The bytes of the receiver are assumed to be ordered as required by the local node.

The `architecture` parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the `getOSPlatform` method of the `Node` class.

The architecture can be one of the `Node` class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>

**setDate**

**Signature**

```java
setDate(day: Integer; month: Integer; year: Integer) updating;
```

The `setDate` method of the `Date` primitive type sets the receiver to a specific date, as shown in the following example.

```java
testDateSet();
vars
date : Date;
begin
date.setDate(31, 08, 1);
write date;    // Outputs 31 August 2001
end;
```

The parameters are the integer values for day, month, and year.

**Note** When you specify a value for the `year` parameter in the range `0` through `99`, the year is assumed to be in the current century. If you want an absolute date in this range, use the `Date::setDateYearAbsolute` method.
**setDateYearAbsolute**

**Signature**

```
setDateYearAbsolute(day: Integer;
month: Integer;
year: Integer) updating;
```

The `setDateYearAbsolute` method of the `Date` primitive type sets the receiver to a specific absolute date. Use this method to specify a date in any year.

The integer values that you define in the `day`, `month`, and `year` variables are initialized when your method is invoked.

The following example shows the use of the `setDateYearAbsolute` method.

```
testAbsoluteDateSet();
vars
    date : Date;
begin
    date.setDateYearAbsolute(31, 08, 1);
    write date;     // Outputs 31 August 0001
end;
```

See also the `Date::setDate` method.

**shortDayNameWithLcid**

**Signature**

```
shortDayNameWithLcid(lcid: Integer): String;
```

The `shortDayNameWithLcid` method of the `Date` primitive type returns a string containing the short name of the week day from the locale specified in the `lcid` parameter for the receiver date.

If the value of the `lcid` parameter is zero (0), the short day name is obtained from the current locale. If the date is null or invalid, an exception is raised.

If you do not define the `EnhancedLocaleSupport` parameter in the `[JadeEnvironment]` section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**shortFormat**

**Signature**

```
shortFormat(): String;
```

The `shortFormat` method of the `Date` primitive type returns the receiver as a string formatted in the short date format. The following example shows the use of the `shortFormat` method.

```
testDateShort();
vars
    date : Date;
begin
    write "The date is " & date.shortFormat;
end;
```

The output from this example depends on the short format defined for the current locale; for example, it may write

The date is 31/8/99.
Notes  You can use the defineShortDateFormat method of the DateFormat class if you want to create your own transient format objects and define a short date format that dynamically overrides the format for the locale at runtime. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

The shortFormat method returns ""invalid"" for dates before 1601 or after 30827.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled.

Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client. For example, if the locale of your application server is set to English (United Kingdom), which has a default short date format of dd/MM/yyyy, and it has been overridden with a short date format of yyyy-MM-dd, this is returned in the default dd/MM/yyyy format.

shortMonthNameWithLcid

Signature  shortMonthNameWithLcid(lcid: Integer): String;

The shortMonthNameWithLcid method of the Date primitive type returns a string containing the short name of the month from the locale specified in the lcId parameter for the receiver date.

If the value of the lcId parameter is zero (0), the short month name is obtained from the current locale. If the date is null or invalid, an exception is raised.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

subtract

Signature  subtract(date: Date): TimeStampInterval;

The subtract method of the Date primitive type returns the interval between the receiver and the date parameter as a TimeStampInterval value.

The following example shows the use of the subtract method.

```java
vars
    today, tomorrow : Date;
begin
    tomorrow := today + 1;
    write tomorrow.subtract(today); // Outputs "1:00:00:00.000"
end;
```

Caution  The TimeStampInterval value that is returned does not take daylight saving into account.
userFormat

**Signature**  
userFormat(fmt: DateFormatter): String;

The `userFormat` method of the *Date* primitive type returns a string containing the receiver in the supplied date format.

To define your date formats, use the Schema menu *Format* command from the Schema Browser.

**Notes**  
When you use a format in a JADE method, prefix your user date format name with a dollar sign ($); for example, `userFormat($MyDate)`.

You can use the `defineLongDateFormat` or `defineShortDateFormat` method of the *DateFormat* class if you want to create your own transient format objects and define a long or short date format that dynamically overrides the appropriate format for the locale at run time. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

In the Windows Environment, the `userFormat` method returns """invalid""" for dates before 1601 or after 30827.

userLongFormatAndLcid

**Signature**  
userLongFormatAndLcid(fmt: DateFormatter; 
  lcid: Integer): String;

The `userLongFormatAndLcid` method of the *Date* primitive type returns a string containing the receiver in the long date format specified for the `fmt` parameter using the locale specified in the `lcid` parameter.

If the value of the `fmt` parameter is null, the long date format of the locale specified in the `lcid` parameter is returned. If the value of the `lcid` parameter is zero (0), the long date format of the current locale is returned.

This method is the same as the `userLongFormatPicAndLcid` method except that the picture string is taken from the *DateFormat* class *longFormat* property. For more details and examples of valid date matches, see the `userLongFormatPicAndLcid` method.

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userLongFormatPicAndLcid

**Signature**  
userLongFormatPicAndLcid(pic: String; 
  lcid: Integer): String;

The `userLongFormatPicAndLcid` method of the *Date* primitive type returns a string containing the receiver in the long date format picture specified for the `pic` parameter using the locale specified in the `lcid` parameter. For examples of `pic` values, see the `parseLongWithPicAndLcid` method.
If the value of the \texttt{pic} parameter is null, the long date format picture of the locale specified in the \texttt{lcid} parameter is returned. If the value of the \texttt{lcid} parameter is zero (0), the long date format picture of the current locale is returned. If the locale is Thai, the year is assumed to include the Thai year offset (that is, 543), which is added to the year included in the returned string.

\textbf{Note} The output text string is a localized Gregorian version of the receiver date. Other calendars such as Hebrew Lunar are not supported.

If you do not define the \texttt{EnhancedLocaleSupport} parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to \texttt{false}, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

\textbf{userShortFormatAndLcid}

\textbf{Signature} \hspace{1em} \texttt{userShortFormatAndLcid(fmt: \texttt{DateFormat}; \newline \hspace{1em} lcid: \texttt{Integer}): \texttt{String;}}

The \texttt{userShortFormatAndLcid} method of the \texttt{Date} primitive type returns a string containing the receiver in the short date format specified for the \texttt{fmt} parameter of the locale specified in the \texttt{lcid} parameter.

If the value of the \texttt{fmt} parameter is null, the short date format of the locale specified in the \texttt{lcid} parameter is returned. If the value of the \texttt{lcid} parameter is zero (0), the short date format of the current locale is returned.

This method is the same as the \texttt{userShortFormatPicAndLcid} method except that the picture string is taken from the \texttt{DateFormat} class \texttt{shortFormat} property. For more details and examples of valid date matches, see the \texttt{userShortFormatPicAndLcid} method.

If you do not define the \texttt{EnhancedLocaleSupport} parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to \texttt{false}, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

\textbf{userShortFormatPicAndLcid}

\textbf{Signature} \hspace{1em} \texttt{userShortFormatPicAndLcid(pic: \texttt{String}; \newline \hspace{1em} lcid: \texttt{Integer}): \texttt{String;}}

The \texttt{userShortFormatPicAndLcid} method of the \texttt{Date} primitive type returns a string containing the receiver in the short date format picture specified for the \texttt{pic} parameter of the locale specified in the \texttt{lcid} parameter. For examples of \texttt{pic} values, see the \texttt{parseShortWithPicAndLcid} method.

If the value of the \texttt{pic} parameter is null, the short date format picture of the locale specified in the \texttt{lcid} parameter is returned. If the value of the \texttt{lcid} parameter is zero (0), the short date format picture of the current locale is returned.

\textbf{Notes} The output text string is a localized Gregorian version of the receiver date. Other calendars such as Hebrew Lunar are not supported.

If the locale is Thai, the Thai year offset (that is, 543) is added to the year included in the returned string.
If you do not define the `EnhancedLocaleSupport` parameter in the `[JadeEnvironment]` section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**year**

**Signature**

`year(): Integer;`

The `year` method of the `Date` primitive type returns the year (for the date value of the receiver) as an integer; for example, `1999`.

The following example shows the use of the `year` method.

```java
if text <> "" then
    calendar.changeType := calendar.ChangeType_Day;
    calendar.date.setDate(text.Integer, calendar.date.month, calendar.date.year);
endif;
```
Decimal Type

Use the **Decimal** primitive type to define a variable with a fixed-precision and decimal point; for example, a monetary value such as a bank balance.

**Note** You must specify a decimal descriptor for the integer length of the decimal variable; for example, code `d : Decimal[4];` to specify a length of 4. You can optionally specify the number of decimal places for the decimal variable; for example, code `d : Decimal[4,3];` to specify that the variable is to be to three decimal places. (If the number of decimal places is not specified, it is assumed to be zero.)

The value of the integer length must be in the range 1 through 23. The number of decimal places must be equal to or less than the length value of the decimal descriptor.

As the decimal value for zero (0) has no fixed precision or scale factor, you can assign this value to any **Decimal** value. When this value is output (for example, by a `write` instruction), the scale factor of the property or variable defines the number of decimal places that are displayed. Null decimal values are initialized with a value of zero (0).

When performing decimal arithmetic, only the final assignment result is rounded. For example:

```plaintext
vars
  a    : Decimal[12,4];
  b, tot : Decimal[12,2];
begin
  a := 4.9350;
  b := a;
  tot := tot + a + a;
write b;       // outputs 4.94, being the final assignment result
write tot;     // outputs 9.87
end;
```

Any intermediate result keeps as much precision as possible, to minimize the overall rounding loss. To force the rounding or truncation of intermediate results, use the `roundedTo` or `truncatedTo` method, respectively. You can use the **JadeEditMask** class and **TextBox** class `getTextAsDecimal` and `setTextFromDecimal` methods to handle locale formatting for numeric fields.

For details about the methods defined in the **Decimal** primitive type, see "Decimal Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the JADE Developer's Reference.

**Decimal Methods**

The methods defined in the **Decimal** primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs</code></td>
<td>Returns the absolute value of the receiver</td>
</tr>
<tr>
<td><code>asBinary</code></td>
<td>Returns the <strong>Binary</strong> representation of the receiver</td>
</tr>
<tr>
<td><code>asDecimal</code></td>
<td>Returns the receiver as a <strong>Decimal</strong> with the specified precision and decimal places</td>
</tr>
<tr>
<td><code>currencyFormat</code></td>
<td>Returns the receiver as a string in the currency format of the current locale</td>
</tr>
<tr>
<td><code>display</code></td>
<td>Returns a string containing the receiver</td>
</tr>
</tbody>
</table>
## Decimal Type

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getDeclaredPrecision</code></td>
<td>Returns the declared precision (length) of a <code>Decimal</code> variable</td>
</tr>
<tr>
<td><code>getDeclaredScaleFactor</code></td>
<td>Returns the declared number of decimal places of a <code>Decimal</code> variable</td>
</tr>
<tr>
<td><code>numberFormat</code></td>
<td>Returns the receiver as a string in the numeric format of the current locale</td>
</tr>
<tr>
<td><code>parseCurrencyWithCurrentLocale</code></td>
<td>Sets the receiver to the result of parsing a string representing a currency value for the current locale</td>
</tr>
<tr>
<td><code>parseCurrencyWithFmtAndLcid</code></td>
<td>Sets the receiver to the result of parsing a string representing a currency value for the specified format and the specified locale</td>
</tr>
<tr>
<td><code>parseNumberWithCurrentLocale</code></td>
<td>Sets the receiver to the result of parsing a string representing a number for the current locale</td>
</tr>
<tr>
<td><code>parseNumberWithFmtAndLcid</code></td>
<td>Sets the receiver to the result of parsing a string representing a number for the specified format and the specified locale</td>
</tr>
<tr>
<td><code>rounded</code></td>
<td>Returns an integer containing the rounded value of the receiver</td>
</tr>
<tr>
<td><code>rounded64</code></td>
<td>Returns a 64-bit integer containing the rounded value of the receiver</td>
</tr>
<tr>
<td><code>roundedTo</code></td>
<td>Returns the receiver rounded to the specified number of decimal places</td>
</tr>
<tr>
<td><code>setByteOrderLocal</code></td>
<td>Returns a <code>Decimal</code> that has the bytes ordered as required by the local node</td>
</tr>
<tr>
<td><code>setByteOrderRemote</code></td>
<td>Returns a <code>Decimal</code> that has the bytes ordered as required by the specified remote node</td>
</tr>
<tr>
<td><code>truncated</code></td>
<td>Returns an integer containing the truncated value of the receiver</td>
</tr>
<tr>
<td><code>truncated64</code></td>
<td>Returns a 64-bit integer containing the truncated value of the receiver</td>
</tr>
<tr>
<td><code>truncatedTo</code></td>
<td>Returns the receiver truncated to the specified number of decimal places</td>
</tr>
<tr>
<td><code>userCurrencyFormat</code></td>
<td>Returns the receiver as a string in the specified currency format</td>
</tr>
<tr>
<td><code>userCurrencyFormatAndLcid</code></td>
<td>Returns the receiver as a string in the specified currency format for the current locale</td>
</tr>
<tr>
<td><code>userNumberFormat</code></td>
<td>Returns the receiver as a string in the specified number format</td>
</tr>
<tr>
<td><code>userNumberFormatAndLcid</code></td>
<td>Returns the receiver as a string in the specified number format for the specified locale</td>
</tr>
</tbody>
</table>

### abs

**Signature**

```
abs(): Decimal;
```

The `abs` method of the `Decimal` primitive type returns a decimal containing the absolute value of the receiver.

### asBinary

**Signature**

```
asBinary(): Binary;
```

The `asBinary` method of the `Decimal` primitive type returns the `Binary` representation of the receiver.

In situations where the compatibility with earlier releases is required, you should use this method in preference to casting the `Decimal` value to a `Binary` value (that is, `Decimal.Binary`).
Casting a Decimal value to a Binary returns the binary representation of the internal representation of the decimal. This depends on the implementation, which is subject to change.

\[
dec := \text{bin.asDecimal}; \quad // \text{Prefer this to } dec := \text{bin.Decimal;}
\]

To convert the resulting binary value back to a decimal value, use the asDecimal method of the Binary primitive type.

\textbf{asDecimal}

\textbf{Signature} \quad \text{asDecimal}(\text{precision: Integer;}
\quad \text{decimalPlaces: Integer}): \text{Decimal;}

The asDecimal method of the Decimal primitive type returns the receiver as a Decimal with the length and scale factor specified by the values of the \textit{precision} and \textit{decimalPlaces} parameters, respectively.

You can use this method to convert a Decimal value returned by a method to a format with the specified precision and decimal places. If you reduce the number of decimal places, rounding occurs. If the receiver value overflows the specified precision, an exception is raised.

\textbf{currencyFormat}

\textbf{Signature} \quad \text{currencyFormat}(): \text{String;}

The currencyFormat method of the Decimal primitive type returns a string containing the receiver in the currency format defined for the current locale; for example, $123.22 or -123.22$. This can include currency symbols, thousands separators, sign characters, and decimal point characters.

The following examples show the use of the currencyFormat method.

\[
\begin{align*}
// \text{calculate average cost of transactions} \\
\text{tblPortfolio.column} := 2; \\
\text{tblPortfolio.text} := \text{portfolio.averageCost.Decimal.currencyFormat;}
\text{testDecimal();}
\text{vars}
\quad \text{decimalValue} : \text{Decimal} [12,4]; \quad // \text{Define the Decimal variable}
\text{begin}
\quad \text{decimalValue} := 1234.56; \quad // \text{Defines the variable value}
\quad \text{write decimalValue;} \quad // \text{Outputs 1234.5600}
\quad \text{write decimalValue.currencyFormat;} \quad // \text{Outputs $1,234.56}$
\quad \text{decimalValue} := -123456; \quad // \text{Defines the variable value}
\quad \text{write decimalValue;} \quad // \text{Outputs -123456.0000}
\quad \text{write decimalValue.currencyFormat;} \quad // \text{Outputs ($123,456.00$)}
\text{end;}
\]

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

\textbf{display}

\textbf{Signature} \quad \text{display}(): \text{String;}

The display method of the Decimal primitive type returns a string containing the receiver.
getDeclaredPrecision
Signature    getDeclaredPrecision(): Integer;

The *getDeclaredPrecision* method of the *Decimal* primitive type returns the declared length of the receiver; that is, the first value within the brackets ([[]]) defined for the *Decimal* variable.

getDeclaredScaleFactor
Signature    getDeclaredScaleFactor(): Integer;

The *getDeclaredScaleFactor* method of the *Decimal* primitive type returns the declared scale factor, or number of decimal places, of the receiver. This method returns the optional second value within the brackets ([[]]) defined for the *Decimal* variable. (The first value is the precision, or length, of the *Decimal* variable.) If the scale factor was not declared, this method returns zero (0).

numberFormat
Signature    numberFormat(): String;

The *numberFormat* method of the *Decimal* primitive type returns a string containing the receiver in the numeric format defined for the current locale; for example, 07456.357 or 7,456.38. This can include thousands separators, sign characters, and decimal point characters. The following example shows the use of the *numberFormat* method.

```
testDecimal();
vars
decimalValue : Decimal [12,4]; // Define the Decimal variable
begin
decimalValue := 1234.56; // Defines the variable value
write decimalValue; // Outputs 1234.5600
write decimalValue.numberFormat; // Outputs 1234.56
// Defines the variable value
write decimalValue; // Outputs -123456.0000
write decimalValue.numberFormat; // Outputs -123,456.00
end;
```

If you do not define the *EnhancedLocaleSupport* parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to *false*, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseCurrencyWithCurrentLocale
Signature    parseCurrencyWithCurrentLocale(source: String; errOffset: Integer output): Integer updating;

The *parseCurrencyWithCurrentLocale* method of the *Decimal* primitive type parses the string specified in the *source* parameter to ensure that it matches the *Decimal* format of the current locale for currency character sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.
If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output errOffset parameter, and sets the receiver to zero (0).

This is equivalent to calling the parseCurrencyWithFmtAndLcid method, passing null in the fmt parameter and zero (0) in the lcid parameter.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseCurrencyWithFmtAndLcid**

**Signature**  
parseCurrencyWithFmtAndLcid(source: String; fmt: CurrencyFormat; lcid: Integer; errOffset: Integer output): Integer updating;

The parseCurrencyWithFmtAndLcid method of the Decimal primitive type parses the string specified in the source parameter using the specified format and locale, to ensure that it matches the format specified in the fmt parameter for currency character sequence, currency position, sign character sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the fmt parameter is null, the settings for the locale specified in the lcid parameter are used. If the value of the lcid parameter is zero (0), the settings of the current locale are used.

If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output errOffset parameter, and sets the receiver to zero (0).

The currency character sequence is optional but if it is included in the source, it must be correctly positioned as defined by the NumberFormat class negativeFormat property and the CurrencyFormat class positiveFormat property.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

A space included in the sign and currency character sequence is optional.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The decimal descriptor of the receiver adds restrictions to the permitted value in the source parameter string; for example, a descriptor of [8,2] allows the value to have up to eight significant digits, with at most two significant digits following the decimal point and six preceding it. Leading zeros before the decimal point and trailing zeros after the decimal point are ignored. The following values are valid:

- "123456.78"
- "1"
- "0000001.2300"
- ".01"
All significant digits in the source parameter string must be able to be stored in the receiver so that they are all shown if the receiver is converted back to a string.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseNumberWithCurrentLocale**

**Signature**

```
parseNumberWithCurrentLocale(source: String;
   errOffset: Integer output): Integer updating;
```

The `parseNumberWithCurrentLocale` method of the `Decimal` primitive type parses the string specified in the source parameter to ensure that it matches the `Decimal` format of the current locale for sign character sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to zero (0).

This is equivalent to calling the `parseNumberWithFmtAndLcid` method, passing null in the `fmt` parameter and zero (0) in the `lcid` parameter.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseNumberWithFmtAndLcid**

**Signature**

```
parseNumberWithFmtAndLcid(source: String;
   fmt: NumberFormat;
   lcid: Integer;
   errOffset: Integer output): Integer updating;
```

The `parseNumberWithFmtAndLcid` method of the `Decimal` primitive type parses the string specified in the source parameter using the specified format and locale, to ensure that it matches the format specified in the `fmt` parameter for sign character sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the `fmt` parameter is null, the settings for the locale specified in the `lcid` parameter are used. If the value of the `lcid` parameter is zero (0), the settings of the current locale are used.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to the invalid value.

The negative sign sequence is optional but if it is included in the source, it must be correctly positioned. A space included in the sign sequence is optional. There is no positive sign sequence.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).
If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The decimal descriptor of the receiver adds restrictions to the permitted value in the source parameter string; for example, a descriptor of [8,2] allows the value to have up to eight significant digits, with at most two significant digits following the decimal point and six preceding it. Leading zeros before the decimal point and trailing zeros after the decimal point are ignored.

The following values are valid.

- "123456.78"
- "1"
- "0000001.2300"
- ".01"

All significant digits in the source parameter string must be able to be stored in the receiver so that they are all shown if the receiver is converted back to a string.

If you do not define the [EnhancedLocaleSupport] parameter in the JadeEnvironment section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

### rounded

**Signature**

rounded(): Integer;

The **rounded** method of the **Decimal** primitive type returns an integer containing the rounded value of the receiver.

The following code fragment shows the use of the **rounded** method.

```java
write (6.4).Decimal.rounded;  // outputs 6 [0,1,2,3,4 are rounded down]
write (6.5).Decimal.rounded;  // outputs 7 [5,6,7,8,9 are rounded up]
write (-6.4).Decimal.rounded; // outputs -6 [0,1,2,3,4 are rounded up]
write (-6.5).Decimal.rounded; // outputs -7 [5,6,7,8,9 are rounded down]
```

### rounded64

**Signature**

rounded64(): Integer64;

The **rounded64** method of the **Decimal** primitive type returns a signed 64-bit integer containing the rounded value of the receiver.

The following code fragment shows the use of the **rounded64** method.

```java
write (6.4).Decimal.rounded64;  // outputs 6 [0,1,2,3,4 are rounded down]
write (6.5).Decimal.rounded64;  // outputs 7 [5,6,7,8,9 are rounded up]
write (-6.4).Decimal.rounded64; // outputs -6 [0,1,2,3,4 are rounded up]
write (-6.5).Decimal.rounded64; // outputs -7 [5,6,7,8,9 are rounded down]
```
roundedTo

Signature roundedTo(decimalPlaces: Integer): Decimal;

The roundedTo method of the Decimal primitive type returns the receiver rounded to the number of decimal places specified in the decimalPlaces parameter.

The following code fragment shows the use of the roundedTo method.

```plaintext
write (3.64).Decimal.roundedTo(1); // outputs 3.6 [0,1,2,3,4 are rounded down]
write (3.65).Decimal.roundedTo(1); // outputs 3.7 [5,6,7,8,9 are rounded up]

write (-3.64).Decimal.roundedTo(1); // outputs -3.6 [0,1,2,3,4 are rounded up]
write (-3.65).Decimal.roundedTo(1); // outputs -3.7 [5,6,7,8,9 are rounded down]
```

setByteOrderLocal

Signature setByteOrderLocal(architecture: Integer): Decimal;

The setByteOrderLocal method of the Decimal primitive type returns a decimal that has the bytes ordered as required by the local node. The bytes of the receiver are assumed to be ordered as indicated by the architecture parameter.

The architecture parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the getOSPlatform method of the Node class.

The architecture can be one of the Node class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>

Note This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setByteOrderRemote

Signature setByteOrderRemote(architecture: Integer): Decimal;

The setByteOrderRemote method of the Decimal primitive type returns a decimal that has the bytes ordered as required by the remote node indicated by the architecture parameter.

The bytes of the receiver are assumed to be ordered as required by the local node.

The architecture parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the getOSPlatform method of the Node class.
The architecture can be one of the **Node** class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>

**Note** This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

### truncated

**Signature**  
truncated(): Integer;

The **truncated** method of the **Decimal** primitive type returns an integer containing the truncated value of the receiver.

**Caution** As the **truncated** method returns an integer value, an integer overflow situation occurs when the returned integer value is greater than the value of the global constant **Max_Integer**, which is the limit for the **Integer** type. To safeguard against this when truncating a large decimal value, use the **Decimal** type **truncated64** method with a parameter of 0 decimal places. Alternatively use the **Decimal** type **truncated64** method.

The following example shows the use of the **truncated** method.

```plaintext
testDecimal();
vars
decimalValue : Decimal [12,4];
begin
decimalValue := 340.5678;       // Defines the variable value
decimalValue := (decimalValue / 20).truncated; // Outputs 17.0000
write decimalValue;
end;
```

### truncated64

**Signature**  
truncated64(): Integer64;

The **truncated64** method of the **Decimal** primitive type returns a signed 64-bit integer containing the truncated value of the receiver.

The following example shows the use of the **truncated** method.

```plaintext
testDecimal();
vars
decimalValue : Decimal [12,4];
begin
decimalValue := 340.5678;       // Defines the variable value
decimalValue := (decimalValue / 20).truncated64;  
```

truncated64(): Integer64;

The truncated64 method of the Decimal primitive type returns a signed 64-bit integer containing the truncated value of the receiver.

The following example shows the use of the truncated method.

```plaintext
testDecimal();
vars
decimalValue : Decimal [12,4];
begin
decimalValue := 340.5678;       // Defines the variable value
decimalValue := (decimalValue / 20).truncated64;
```
Encyclopaedia of Primitive Types

Decimal Type

```
write decimalValue; // Outputs 17.0000
end;
```

**truncatedTo**

**Signature**  
truncatedTo(decimalPlaces: Integer): Decimal;

The **truncatedTo** method of the **Decimal** primitive type returns the receiver truncated to the number of decimal places specified in the **decimalPlaces** parameter.

The following example shows the use of the **truncatedTo** method.

```
testDecimal();
vars
  decimalValue : Decimal [12,4];
begin
  decimalValue := 340.56789; // Defines the variable value
  decimalValue := (decimalValue/20).truncatedTo(2);
  write decimalValue; // Outputs 17.0200
end;
```

**userCurrencyFormat**

**Signature**  
userCurrencyFormat(fmt: CurrencyFormat): String;

The **userCurrencyFormat** method of the **Decimal** primitive type returns a string containing the receiver in the currency format specified in the **fmt** parameter.

To define your currency formats, use the Schema menu **Format** command from the Schema Browser.

The code fragment in the following example shows the use of the **userCurrencyFormat** method.

```
lblBank.caption := app.crntInvestor.cash.userCurrencyFormat($DollarsCents);
lblWorth.caption := (totalWorth).userCurrencyFormat($DollarsCents);
```

**Notes**  
When you use a format in a JADE method, prefix your user currency format name with a dollar sign ($); for example, **userCurrencyFormat($MyCurrency)**.

You can use the **defineCurrencyFormat** method of the **CurrencyFormat** class if you want to create your own transient format objects and define a currency format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
userCurrencyFormatAndLcid

Signature: userCurrencyFormatAndLcid(fmt: CurrencyFormat; lcid: Integer): String;

The userCurrencyFormatAndLcid method of the Decimal primitive type returns a string containing the receiver in the currency format and locale specified in the fmt parameter and lcid parameter, respectively. If the value of the fmt parameter is null, the settings for the locale specified in the lcid parameter are used. If the value of the lcid parameter is zero (0), the settings of the current locale are used.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormat

Signature: userNumberFormat(fmt: NumberFormat): String;

The userNumberFormat method of the Decimal primitive type returns a string containing the receiver in the number format specified in the fmt parameter. To define your numeric formats, use the Schema menu Format command from the Schema Browser.

Notes: When you use a format in a JADE method, prefix your user number format name with a dollar sign ($); for example, userNumberFormat($MyNumber).

You can use the defineNumberFormat method of the NumberFormat class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at runtime. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The code fragment in the following example shows the use of the userNumberFormat method.

```java
if totalWorth = 0 then
tblPortfolio.text := "0.00";
else
  tblPortfolio.text := (portfolio.myCompany.currentPrice *
                        portfolio.shares.Decimal * Percent
                        / totalWorth).Decimal.userNumberFormat($Percent);
endif;
```

userNumberFormatAndLcid

Signature: userNumberFormatAndLcid(fmt: NumberFormat; lcid: Integer): String;

The userNumberFormatAndLcid method of the Decimal primitive type returns a string containing the receiver in the number format and locale specified in the fmt parameter and lcid parameter, respectively.
If the value of the `fmt` parameter is null, the settings for the locale specified in the `lcid` parameter are used. If the value of the `lcid` parameter is zero (0), the settings of the current locale are used.

If you do not define the `EnhancedLocaleSupport` parameter in the `[JadeEnvironment]` section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
Integer Type

The **Integer** primitive type represents the set of positive and negative whole numbers in the range -2,147,483,648 through 2,147,483,647. Any value of the **Integer** primitive type is therefore a whole number.

JADE defines a number of arithmetic operations that take integer operators and return integer results, as listed in the following table.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td>div</td>
<td>Integer division (division with truncation; for example, 7 div 3 = 2)</td>
</tr>
<tr>
<td>^</td>
<td>Exponentiation (for example, i ^ 3 is i cubed)</td>
</tr>
<tr>
<td>mod</td>
<td>Modulus (remainder after integer division)</td>
</tr>
</tbody>
</table>

These are binary (or dyadic) infix operators; that is, they are used with two operands written on each side of the operator (for example, a+b). However, the + operator and - operator can also be used as unary (or monadic) prefix operators, as listed in the following table:

<table>
<thead>
<tr>
<th>Unary Prefix Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+a</td>
<td>Sign identify</td>
</tr>
<tr>
<td>-a</td>
<td>Sign inversion</td>
</tr>
</tbody>
</table>

The `div` operator (integer division) performs division with truncation; for example, 7 div 3 = 2.

An integer variable can contain any *whole* number in the range -2,147,483,648 through 2,147,483,647.

The following table lists valid operations for the **Integer** primitive type.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expression Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer-expression + date-expression</td>
<td>(date)</td>
</tr>
<tr>
<td>integer-expression + time-expression</td>
<td>(time)</td>
</tr>
</tbody>
</table>

You can use the **JadeEditMask** class and **TextBox** class **getTextAsInteger** and **setTextFromInteger** methods to handle locale formatting for numeric fields.

For details about the methods defined in the **Integer** primitive type, see "**Integer Methods**", in the following subsection. For details about converting primitive types, see "**Converting Primitive Types**", in Chapter 1 of the **JADE Developer's Reference**.
# Integer Methods

The methods defined in the `Integer` primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>Returns the absolute value of the receiver</td>
</tr>
<tr>
<td>bitAnd</td>
<td>Returns an integer representing the receiver bits ANDed with the argument</td>
</tr>
<tr>
<td>bitNot</td>
<td>Returns an integer whose bit values are the inverse of the bit values of the receiver</td>
</tr>
<tr>
<td>bitOr</td>
<td>Returns an integer representing the receiver bits ORed with the argument</td>
</tr>
<tr>
<td>bitXor</td>
<td>Returns an integer representing the receiver bits XORed with the argument</td>
</tr>
<tr>
<td>display</td>
<td>Returns the receiver as a string</td>
</tr>
<tr>
<td>isEven</td>
<td>Returns <code>true</code> if the receiver represents an even number; otherwise <code>false</code></td>
</tr>
<tr>
<td>isOdd</td>
<td>Returns <code>true</code> if the receiver represents an odd number; otherwise <code>false</code></td>
</tr>
<tr>
<td>max</td>
<td>Returns the larger value of the receiver and a specified <code>Integer</code></td>
</tr>
<tr>
<td>min</td>
<td>Returns the lesser value of the receiver and a specified <code>Integer</code></td>
</tr>
<tr>
<td>numberFormat</td>
<td>Returns a string in the number format of the current locale</td>
</tr>
<tr>
<td>padLeadingWith</td>
<td>Returns a copy of the receiver padded to the specified length with a leading character</td>
</tr>
<tr>
<td>parseCurrencyWithCurrentLocale</td>
<td>Sets the receiver to the result of parsing a string representing a currency value for the current locale</td>
</tr>
<tr>
<td>parseCurrencyWithFmtAndLcid</td>
<td>Sets the receiver to the result of parsing a string representing a currency value for the specified format and the specified locale</td>
</tr>
<tr>
<td>parseNumberWithCurrentLocale</td>
<td>Sets the receiver to the result of parsing a string representing a number for the current locale</td>
</tr>
<tr>
<td>parseNumberWithFmtAndLcid</td>
<td>Sets the receiver to the result of parsing a string representing a number for the specified format and the specified locale</td>
</tr>
<tr>
<td>setByteOrderLocal</td>
<td>Returns an integer that has the bytes ordered as required by the local node</td>
</tr>
<tr>
<td>setByteOrderRemote</td>
<td>Returns an integer that has the bytes ordered as required by a specified remote node</td>
</tr>
<tr>
<td>userCurrencyFormat</td>
<td>Returns the receiver as a string in the specified currency format</td>
</tr>
<tr>
<td>userCurrencyFormatAndLcid</td>
<td>Returns the receiver as a string in the specified currency format for the specified locale</td>
</tr>
<tr>
<td>userNumberFormat</td>
<td>Returns the receiver as a string in the specified number format</td>
</tr>
<tr>
<td>userNumberFormatAndLcid</td>
<td>Returns the receiver as a string in the specified number format for the specified locale</td>
</tr>
</tbody>
</table>
### abs

**Signature**  
`abs() : Integer;`

The `abs` method of the `Integer` primitive type returns an integer containing the absolute value of the receiver.

### bitAnd

**Signature**  
`bitAnd(op: Integer) : Integer;`

The `bitAnd` method of the `Integer` primitive type compares each bit in the receiver with the corresponding bit in the `op` parameter, and returns an integer representing the receiver bits ANDed with the argument.

The generated return values are listed in the following table.

<table>
<thead>
<tr>
<th>Bits in Receiver and Operand</th>
<th>Corresponding Bit in Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both bits are 1</td>
<td>1</td>
</tr>
<tr>
<td>One or both bits are not 1</td>
<td>0</td>
</tr>
</tbody>
</table>

The following example shows the use of the `bitAnd` method.

```pascal
vars  
  platform : Integer;  
  version : String;  
  architecture : Integer;
begin  
  platform := node.getOSPlatform(version, architecture);
  if platform.bitAnd(Node.OSWindows) <> 0 then  
    // operating system is Windows family (10, 8, 7, 2008, or  
    // Vista)
    if platform = Node.OSWindowsHome then  
      // version is an older version of Windows (unsupported)  
      return 'Windows (unsupported) ' & version;
    endif;
    if platform = Node.OSWindowsEnterprise then  
      // version is Windows 10, Windows 8, Windows 7, Windows  
      return 'Windows ' & version;
    endif;
    if platform = Node.OSWindowsMobile then  
      // version is Windows CE  
      return 'Windows CE ' & version;
    endif;
  endif;
  return '*' Unknown platform: ' & platform.String & ' version: ' &  
    version;
end;
```

### bitNot

**Signature**  
`bitNot() : Integer;`

The `bitNot` method of the `Integer` primitive type returns an integer whose bit values are the inverse of the bit values of the receiver.
The generated return values are listed in the following table.

<table>
<thead>
<tr>
<th>Bits in Receiver</th>
<th>Corresponding Bit in Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit is not 1</td>
<td>1</td>
</tr>
<tr>
<td>Bit is 1</td>
<td>0</td>
</tr>
</tbody>
</table>

**bitOr**

**Signature**  
`bitOr(op: Integer): Integer;`

The `bitOr` method of the `Integer` primitive type compares each bit in the receiver with the corresponding bit in the `op` parameter, and returns an integer representing the receiver bits ORed with the argument.

The generated return values are listed in the following table.

<table>
<thead>
<tr>
<th>Bits in Receiver and Operand</th>
<th>Corresponding Bit in Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>One or both bits are 1</td>
<td>1</td>
</tr>
<tr>
<td>Neither bit is 1</td>
<td>0</td>
</tr>
</tbody>
</table>

The code fragment in the following example shows the use of the `bitOr` method.

```plaintext
constants
    BitFlagNone = #00;
    BitFlag1   = #01;
vars
    int : Integer;
begin
    int := BitFlagNone;
    // set bit flag 1
    int := int.bitOr(BitFlag1);
    // test that bit flag 1 is set
    if int.bitAnd(BitFlag1) <> 0 then
        write "flag 1 is set";
    endif;
end;
```

**bitXor**

**Signature**  
`bitXor(op: Integer): Integer;`

The `bitXor` method of the `Integer` primitive type compares each bit in the receiver with the corresponding bit specified in the `op` parameter, and returns an integer representing the receiver bits XORed with the argument.

The generated return values are listed in the following table.

<table>
<thead>
<tr>
<th>Bits in Receiver and Operand</th>
<th>Corresponding Bit in Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>The bits are complementary</td>
<td>1</td>
</tr>
<tr>
<td>The bits are not complementary</td>
<td>0</td>
</tr>
</tbody>
</table>
**display**

**Signature**  
display(): String;

The display method of the Integer primitive type returns a string containing the receiver.

**isEven**

**Signature**  
isEven(): Boolean;

The isEven method of the Integer primitive type returns true if the receiver represents an even number or it returns false if it does not.

The following example shows the use of the isEven method.

```plaintext
setColor(prod: Product) updating;
begin
  if prod.isNew then
    listProducts.itemForeColor[listProducts.newIndex] := Red;
  else
    listProducts.itemForeColor[listProducts.newIndex] := Gray;
  endif;
  if listProducts.newIndex.isEven then
    listProducts.itemBackColor[listProducts.newIndex] := LightYellow;
  endif;
end;
```

**isOdd**

**Signature**  
isOdd(): Boolean;

The isOdd method of the Integer primitive type returns true if the receiver represents an odd number; otherwise, it returns false.

The code fragment in the following example shows the use of the isOdd method.

```plaintext
if sides.isOdd then
  selectedRectangle.left := (maxWidth - defaultWidth - defaultSpacing) / 2 + 10;
  selectedRectangle.top := (sides - 1) * (defaultHeight + defaultSpacing) / 2 + 10;
else
  selectedRectangle.left := maxWidth / 2 + 10;
  selectedRectangle.top := maxHeight / 2 - defaultHeight - defaultSpacing + 10;
endif;
```

**max**

**Signature**  
max(int: Integer): Integer;

The max method of the Integer primitive type returns the larger value of the receiver and the int parameter.

If the value of the receiver is greater than the value of the int parameter, the value of the receiver is returned. If the value of the receiver is less than or equal to the value of the int parameter, the value of int is returned.
**min**

**Signature** \( \text{min}(\text{int}: \text{Integer}) : \text{Integer}; \)

The `min` method of the `Integer` primitive type returns the lesser value of the receiver and the `int` parameter.

If the value of the receiver is less than the value of the `int` parameter, the value of the receiver is returned. If the value of the receiver is greater than or equal to the value of the `int` parameter, the value of `int` is returned.

**numberFormat**

**Signature** `numberFormat() : String;`

The `numberFormat` method of the `Integer` primitive type returns a string in the number format defined for the current locale; for example, `-7456.000` or `7,456`. This can include thousands separators, sign characters, and decimal point characters.

The following example shows the use of the `numberFormat` method.

```pascal
  testInteger();
  vars
    str : String;
    int : Integer;
  begin
    int := -01234567890;
    write int;    // Outputs -1234567890
    str := int.numberFormat;
    write str;    // Outputs -1,234,567,890.00
  end;
```

If you do not define the `EnhancedLocaleSupport` parameter in the `[JadeEnvironment]` section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**padLeadingWith**

**Signature** `padLeadingWith(char: Character; max: Integer) : String;`

The `padLeadingWith` method of the `Integer` primitive type returns a string of the length specified in the `max` parameter, consisting of the receiving string padded with the leading character specified in the `char` parameter. If the string is equal to or longer than the value specified in the `max` parameter, it is not truncated but the whole string is returned.

The following example shows the use of the `padLeadingWith` method.

```pascal
  constants
    PAD_CHARACTER = 'x';
  vars
    int : Integer;
    str : String;
  begin
    int := -012345;
    str := int.padLeadingWith('w', 15) & ' 678 Sesame St.';
```
write str; // Outputs wwwwwwww-12345 678 Sesame St.
str := int.padLeadingWith('a', 2);
write str; // Outputs -12345
str := int.padLeadingWith(PAD_CHARACTER, 10);
write str; // Outputs xxxx-12345
end;

**parseCurrencyWithCurrentLocale**

**Signature**

```java
parseCurrencyWithCurrentLocale(source: String;
errOffset: Integer output)
  : Integer updating;
```

The `parseCurrencyWithCurrentLocale` method of the **Integer** primitive type parses the string specified in the **source** parameter to ensure that it matches the **Integer** format of the current locale for currency character sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **source** parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to zero (0).

This is equivalent to calling the `parseCurrencyWithFmtAndLcid` method, passing null in the **fmt** parameter and zero (0) in the **lcid** parameter.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseCurrencyWithFmtAndLcid**

**Signature**

```java
parseCurrencyWithFmtAndLcid(source: String;
fmt: CurrencyFormat;
lcid: Integer;
errOffset: Integer output): Integer updating;
```

The `parseCurrencyWithFmtAndLcid` method of the **Integer** primitive type parses the string specified in the **source** parameter using the specified format and locale, to ensure that it matches the format specified in the **fmt** parameter for currency character sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the **fmt** parameter is null, the settings for the locale specified in the **lcid** parameter are used. If the value of the **lcid** parameter is zero (0), the settings of the current locale are used.

If the value of the **source** parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output **errOffset** parameter, and sets the receiver to the invalid value.

The currency character sequence is optional but if it is included in the source, it must be correctly positioned as defined by the **NumberFormat** class **negativeFormat** property and the **CurrencyFormat** class **positiveFormat** property.
Thousands separator character sequences are optional but if they are included in the source, each one must have
at least one digit preceding and following it, and must occur before the decimal point (if any).

A space included in the sign and currency character sequence is optional.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be
native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The value of the source parameter text can include a decimal point and decimal digits, but they must all be zero
so that rounding or truncation is not required to store the value in the Integer variable; for example:

- "100", "100.", "100.0", "100.00", and "100.000" are accepted as valid and equal.
- "100.01" and "100.99" are rejected, as the value cannot be stored accurately in an Integer primitive type.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE
initialization file on the database node or you set it to false, inconsistent results could be returned to the
application server when running in JADE thin client mode and there are locale overrides, as all overrides on the
application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on
the application server, based on the locale of the corresponding presentation client.

**parseNumberWithCurrentLocale**

**Signature**

```java
parseNumberWithCurrentLocale(source: String; 
errOffset: Integer output): Integer updating;
```

The `parseNumberWithCurrentLocale` method of the `Integer` primitive type parses the string specified in the
source parameter to ensure that it matches the `Integer` format of the current locale for sign character sequence,
sign position, thousands separator, decimal point sequence, and character set.

If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to
the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range
1800 through 1869), indicates the first offending character returning its zero-based offset using the output
errOffset parameter, and sets the receiver to the invalid value.

This is equivalent to calling the `parseNumberWithFmtAndLcid` method, passing null in the fmt parameter and
zero (0) in the lcid parameter.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE
initialization file on the database node or you set it to false, inconsistent results could be returned to the
application server when running in JADE thin client mode and there are locale overrides, as all overrides on the
application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on
the application server, based on the locale of the corresponding presentation client.

**parseNumberWithFmtAndLcid**

**Signature**

```java
parseNumberWithFmtAndLcid(source: String; 
fmt: NumberFormat; 
lcid: Integer; 
errOffset: Integer output): Integer updating;
```

The `parseNumberWithFmtAndLcid` method of the `Integer` primitive type parses the string specified in the source
parameter using the specified format and locale, to ensure that it matches the format specified in the fmt
parameter for sign character sequence, sign position, thousands separator, decimal point sequence, and
character set.
If the value of the `fmt` parameter is null, the settings for the locale specified in the `lcid` parameter are used. If the value of the `lcid` parameter is zero (0), the settings of the current locale are used.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to the invalid value.

The negative sign character sequence is optional but if it is included in the source, it must be correctly positioned. A space included in the sign sequence is optional. There is no positive sign sequence.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The value of the `source` parameter text can include a decimal point and decimal digits, but they must all be zero so that rounding or truncation is not required to store the value in the `Integer` variable; for example:

- "100", "100.", "100.0", "100.00", and "100.000" are accepted as valid and equal.
- "100.01" and "100.99" are rejected, as the value cannot be stored accurately in an `Integer` primitive type.

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**setByteOrderLocal**

**Signature**

```java
setByteOrderLocal(architecture: Integer): Integer;
```

The `setByteOrderLocal` method of the `Integer` primitive type returns an integer that has the bytes ordered as required by the local node. The bytes of the receiver are assumed to be ordered as indicated by the `architecture` parameter.

The `architecture` parameter indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the `getOSPlatform` method of the `Node` class.

The architecture can be one of the `Node` class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Architecture_32Big_Endian</code></td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td><code>Architecture_32Little_Endian</code></td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td><code>Architecture_64Big_Endian</code></td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td><code>Architecture_64Little_Endian</code></td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td><code>Architecture_Gui</code></td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
<tr>
<td>No constant; that is, zero (0)</td>
<td>Reorders the bytes from network byte order (a standard for passing binary integers across a TCP/IP connection).</td>
</tr>
</tbody>
</table>
**Note**  This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

**setByteOrderRemote**

**Signature**  
`setByteOrderRemote(architecture: Integer): Integer;`

The `setByteOrderRemote` method of the `Integer` primitive type returns an integer that has the bytes ordered as required by the remote node indicated by the `architecture` parameter. The bytes of the receiver are assumed to be ordered as required by the local node.

The `architecture` parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the `getOSPlatform` method of the `Node` class.

The architecture can be one of the `Node` class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
<tr>
<td>No constant; that is, zero (0)</td>
<td>Reorders the bytes from network byte order (a standard for passing binary integers across a TCP/IP connection)</td>
</tr>
</tbody>
</table>

**Note**  This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

**userCurrencyFormat**

**Signature**  
`userCurrencyFormat(fmt: CurrencyFormat): String;`

The `userCurrencyFormat` method of the `Integer` primitive type returns a string containing the receiver in the supplied currency format.

To define your currency formats, use the Schema menu `Format` command from the Schema Browser.

**Notes**  When you use a format in a JADE method, prefix your user currency format name with a dollar sign ($); for example, `userCurrencyFormat($MyCurrency)`.

You can use the `defineCurrencyFormat` method of the `CurrencyFormat` class if you want to create your own transient format objects and define a currency format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
userCurrencyFormatAndLcid

Signature  userCurrencyFormatAndLcid(fmt: CurrencyFormat; lcid: Integer): String;

The userCurrencyFormatAndLcid method of the Integer primitive type returns a string containing the receiver in the currency format and locale specified in the fmt parameter and lcid parameter, respectively.

If the value of the fmt parameter is null, the settings for the locale specified in the lcid parameter are used. If the value of the lcid parameter is zero (0), the settings of the current locale are used.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormat

Signature  userNumberFormat(fmt: NumberFormat): String;

The userNumberFormat method of the Integer primitive type returns a string containing the receiver in the supplied number format.

To define your numeric formats, use the Schema menu Format command from the Schema Browser.

Notes  When you use a format in a JADE method, prefix your user number format name with a dollar sign ($); for example, userNumberFormat($MyNumber).

You can use the defineNumberFormat method from the NumberFormat class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at runtime. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormatAndLcid

Signature  userNumberFormatAndLcid(fmt: NumberFormat; lcid: Integer): String;

The userNumberFormatAndLcid method of the Integer primitive type returns a string containing the receiver in the number format and locale specified in the fmt parameter and lcid parameter, respectively.

If the value of the fmt parameter is null, the settings for the locale specified in the lcid parameter are used. If the value of the lcid parameter is zero (0), the settings of the current locale are used.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
Integer64 Type

The **Integer64** primitive type represents the set of positive and negative whole numbers in the range -9,223,720,368,547,758.07 through 9,223,720,368,547,758.07. Any value of the **Integer64** primitive type is therefore a whole number.

JADE defines a number of arithmetic operations that take **Integer64** operators and return **Integer64** results, as listed in the following table.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td>div</td>
<td>Integer division (division with truncation; for example, 7 div 3 = 2)</td>
</tr>
<tr>
<td>^</td>
<td>Exponentiation (for example, i ^ 3 is i cubed)</td>
</tr>
<tr>
<td>mod</td>
<td>Modulus (remainder after integer division)</td>
</tr>
</tbody>
</table>

These are binary (or dyadic) infix operators; that is, they are used with operands on both sides of the operator (for example, a+b). However, the + operator and - operator can also be used as unary (or monadic) prefix operators, as listed in the following table.

<table>
<thead>
<tr>
<th>Unary Prefix Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+a</td>
<td>Sign identify</td>
</tr>
<tr>
<td>-a</td>
<td>Sign inversion</td>
</tr>
</tbody>
</table>

The div operator (integer division) performs division with truncation; for example, 7 div 3 = 2.

For details about the methods defined in the **Integer64** primitive type, see "**Integer64 Methods**", in the following subsection. For details about converting primitive types, see "**Converting Primitive Types**", in Chapter 1 of the **JADE Developer’s Reference**.

### Integer64 Methods

The methods defined in the **Integer64** primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>Returns the absolute value of the receiver</td>
</tr>
<tr>
<td>bitAnd</td>
<td>Returns an integer representing the receiver bits ANDed with the argument</td>
</tr>
<tr>
<td>bitNot</td>
<td>Returns an integer whose bit values are the inverse of the bit values of the receiver</td>
</tr>
<tr>
<td>bitOr</td>
<td>Returns an integer representing the receiver bits ORed with the argument</td>
</tr>
<tr>
<td>bitXor</td>
<td>Returns an integer representing the receiver bits XORed with the argument</td>
</tr>
</tbody>
</table>
### Integer64 Type

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display</td>
<td>Returns the receiver as a string</td>
</tr>
<tr>
<td>isEven</td>
<td>Returns <code>true</code> if the receiver represents an even number; otherwise <code>false</code></td>
</tr>
<tr>
<td>isOdd</td>
<td>Returns <code>true</code> if the receiver represents an odd number; otherwise <code>false</code></td>
</tr>
<tr>
<td>max</td>
<td>Returns the larger value of the receiver and a specified <code>Integer64</code></td>
</tr>
<tr>
<td>min</td>
<td>Returns the lesser value of the receiver and a specified <code>Integer64</code></td>
</tr>
<tr>
<td>numberFormat</td>
<td>Returns a string in the number format of the current locale</td>
</tr>
<tr>
<td>padLeadingWith</td>
<td>Returns a copy of the receiver padded to the specified length with a leading character</td>
</tr>
<tr>
<td>parseCurrencyWithCurrentLocale</td>
<td>Sets the receiver to the result of parsing a string representing a currency value for the current locale</td>
</tr>
<tr>
<td>parseCurrencyWithFmtAndLcid</td>
<td>Sets the receiver to the result of parsing a string representing a currency value for the specified format and the specified locale</td>
</tr>
<tr>
<td>parseNumberWithCurrentLocale</td>
<td>Sets the receiver to the result of parsing a string representing a number for the current locale</td>
</tr>
<tr>
<td>parseNumberWithFmtAndLcid</td>
<td>Sets the receiver to the result of parsing a string representing a number for the specified format and the specified locale</td>
</tr>
<tr>
<td>setByteOrderLocal</td>
<td>Returns an integer that has the bytes ordered as required by the local node</td>
</tr>
<tr>
<td>setByteOrderRemote</td>
<td>Returns an integer that has the bytes ordered as required by a specified remote node</td>
</tr>
<tr>
<td>userCurrencyFormat</td>
<td>Returns the receiver as a string in the specified currency format</td>
</tr>
<tr>
<td>userCurrencyFormatAndLcid</td>
<td>Returns the receiver as a string in the specified currency format for the specified locale</td>
</tr>
<tr>
<td>userNumberFormat</td>
<td>Returns the receiver as a string in the specified number format</td>
</tr>
<tr>
<td>userNumberFormatAndLcid</td>
<td>Returns the receiver as a string in the specified number format for the specified locale</td>
</tr>
</tbody>
</table>

### abs

**Signature**

```plaintext
abs(): Integer64;
```

The `abs` method of the `Integer64` primitive type returns an integer containing the absolute value of the receiver.

### bitAnd

**Signature**

```plaintext
bitAnd(op: Integer64): Integer64;
```

The `bitAnd` method of the `Integer64` primitive type compares each bit in the receiver with the corresponding bit in the `op` parameter, and returns an integer representing the receiver bits ANDed with the argument.
The generated return values are listed in the following table.

<table>
<thead>
<tr>
<th>Bits in Receiver and Operand</th>
<th>Corresponding Bit in Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both bits are 1</td>
<td>1</td>
</tr>
<tr>
<td>One or both bits are not 1</td>
<td>0</td>
</tr>
</tbody>
</table>

**bitNot**

**Signature**  
bitNot(): Integer64;

The **bitNot** method of the **Integer64** primitive type returns an integer whose bit values are the inverse of the bit values of the receiver.

The generated return values are listed in the following table.

<table>
<thead>
<tr>
<th>Bits in Receiver</th>
<th>Corresponding Bit in Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit is not 1</td>
<td>1</td>
</tr>
<tr>
<td>Bit is 1</td>
<td>0</td>
</tr>
</tbody>
</table>

**bitOr**

**Signature**  
bitOr(op: Integer64): Integer64;

The **bitOr** method of the **Integer64** primitive type compares each bit in the receiver with the corresponding bit in the **op** parameter, and returns an integer representing the receiver bits ORed with the argument.

The generated return values are listed in the following table.

<table>
<thead>
<tr>
<th>Bits in Receiver and Operand</th>
<th>Corresponding Bit in Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>One or both bits are 1</td>
<td>1</td>
</tr>
<tr>
<td>Neither bit is 1</td>
<td>0</td>
</tr>
</tbody>
</table>

The code fragment in the following example shows the use of the **bitOr** method.

```constant
    BitFlagNone = #00;
    BitFlag1 = #01;

var
    int : Integer64;
begin
    int := BitFlagNone;
    // set bit flag 1
    int := int.bitOr(BitFlag1);
    // test that bit flag 1 is set
    if int.bitAnd(BitFlag1) <> 0 then
        write "flag 1 is set";
    endif;
end;
```
**bitXor**

**Signature**  
`bitXor(op: Integer64): Integer64;`

The `bitXor` method of the `Integer64` primitive type compares each bit in the receiver with the corresponding bit specified in the `op` parameter, and returns an integer representing the receiver bits XORed with the argument.

The generated return values are listed in the following table.

<table>
<thead>
<tr>
<th>Bits in Receiver and Operand</th>
<th>Corresponding Bit in Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>The bits are complementary</td>
<td>1</td>
</tr>
<tr>
<td>The bits are not complementary</td>
<td>0</td>
</tr>
</tbody>
</table>

**display**

**Signature**  
`display(): String;`

The `display` method of the `Integer64` primitive type returns a string containing the receiver.

**isEven**

**Signature**  
`isEven(): Boolean;`

The `isEven` method of the `Integer64` primitive type returns `true` if the receiver represents an even number or it returns `false` if it does not.

**isOdd**

**Signature**  
`isOdd(): Boolean;`

The `isOdd` method of the `Integer64` primitive type returns `true` if the receiver represents an odd number; otherwise, it returns `false`.

**max**

**Signature**  
`max(int: Integer64): Integer64;`

The `max` method of the `Integer64` primitive type returns the larger value of the receiver and the `int` parameter. If the value of the receiver is greater than the value of the `int` parameter, the value of the receiver is returned. If the value of the receiver is less than or equal to the value of the `int` parameter, the value of `int` is returned.

**min**

**Signature**  
`min(int: Integer64): Integer64;`

The `min` method of the `Integer64` primitive type returns the lesser value of the receiver and the `int` parameter. If the value of the receiver is less than the value of the `int` parameter, the value of the receiver is returned. If the value of the receiver is greater than or equal to the value of the `int` parameter, the value of `int` is returned.
numberFormat

**Signature**  
numberFormat(): String;

The `numberFormat` method of the `Integer64` primitive type returns a string in the number format defined for the current locale; for example, `-7456.000` or `7,456`. This can include thousands separators, sign characters, and decimal point characters.

The following example shows the use of the `numberFormat` method.

```plaintext
testInteger();
vars
  str : String;
  int : Integer64;
begin
  int := -01234567890;
  write int; // Outputs -1234567890
  str := int.numberFormat; // Outputs -1,234,567,890.00
  write str;
end;
```

You can use the `defineNumberFormat` method of the `NumberFormat` class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the `EnhancedLocaleSupport` parameter in the `[JadeEnvironment]` section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

padLeadingWith

**Signature**  
padLeadingWith(char: Character;  
max: Integer64): String;

The `padLeadingWith` method of the `Integer64` primitive type returns a string of the length specified in the `max` parameter, consisting of the receiving string padded with the leading character specified in the `char` parameter. If the string is equal to or longer than the value specified in the `max` parameter, it is not truncated but the whole string is returned.

The following example shows the use of the `padLeadingWith` method.

```plaintext
costants
  PAD_CHARACTER = 'x';
vars
  int : Integer64;
  str : String;
begin
  int := -012345;
  str := int.padLeadingWith('w', 15) & ' 678 Sesame St.';
  write str; // Outputs wwwwwww-12345 678 Sesame St.
  str := int.padLeadingWith('a', 2);
  write str; // Outputs -12345
  str := int.padLeadingWith(PAD_CHARACTER, 10);
```
parseCurrencyWithCurrentLocale

Signature  parseCurrencyWithCurrentLocale(source:  String;
         errOffset: Integer output): Integer updating;

The parseCurrencyWithCurrentLocale method of the Integer64 primitive type parses the string specified in the source parameter to ensure that it matches the Integer64 format of the current locale for currency character sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1860 through 1869), indicates the first offending character returning its zero-based offset using the output errOffset parameter, and sets the receiver to zero (0).

This is equivalent to calling the parseCurrencyWithFmtAndLcid method, passing null in the fmt parameter and zero (0) in the lcid parameter.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

parseCurrencyWithFmtAndLcid

Signature  parseCurrencyWithFmtAndLcid(source:  String;
         fmt:   CurrencyFormat;
         lcid:  Integer;
         errOffset: Integer output): Integer updating;

The parseCurrencyWithFmtAndLcid method of the Integer64 primitive type parses the string specified in the source parameter using the specified format and locale, to ensure that it matches the format specified in the fmt parameter for currency character sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the fmt parameter is null, the settings for the locale specified in the lcid parameter are used. If the value of the lcid parameter is zero (0), the settings of the current locale are used.

If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1860 through 1869), indicates the first offending character returning its zero-based offset using the output errOffset parameter, and sets the receiver to the invalid value.

The currency sequence is optional but if it is included in the source, it must be correctly positioned as defined by the NumberFormat class negativeFormat property and the CurrencyFormat class positiveFormat property.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

A space included in the sign and currency character sequence is optional.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.
The value of the source parameter text can include a decimal point and decimal digits, but they must all be zero so that rounding or truncation is not required to store the value in the Integer64 variable; for example:

- "100", "100.", "100.0", and "100.000" are accepted as valid and equal.
- "100.01" and "100.99" are rejected, as the value cannot be stored accurately in an Integer64 primitive type.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseNumberWithCurrentLocale**

Signature: `parseNumberWithCurrentLocale(source: String; errOffset: Integer output): Integer updating;`

The parseNumberWithCurrentLocale method of the Integer64 primitive type parses the string specified in the source parameter to ensure that it matches the Integer64 format of the current locale for sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output errOffset parameter.

This is equivalent to calling the parseNumberWithFmtAndLcid method, passing null in the fmt parameter and zero (0) in the lcid parameter.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseNumberWithFmtAndLcid**

Signature: `parseNumberWithFmtAndLcid(source: String; fmt: NumberFormat; lcid: Integer; errOffset: Integer output): Integer updating;`

The parseNumberWithFmtAndLcid method of the Integer64 primitive type parses the string specified in the source parameter using the specified format and locale, to ensure that it matches the format specified in the fmt parameter for sign character sequence, sign position, thousands separator, decimal point sequence, and character set. If the value of the fmt parameter is null, the settings for the locale specified in the lcid parameter are used. If the value of the lcid parameter is zero (0), the settings of the current locale are used.

If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output errOffset parameter, and sets the receiver to the invalid value.

The negative sign sequence is optional but if it is included in the source, it must be correctly positioned. A space included in the sign sequence is optional. There is no positive sign sequence.
Integer64 Type

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

The value of the source parameter text can include a decimal point and decimal digits, but they must all be zero so that rounding or truncation is not required to store the value in the Integer64 variable; for example:

- "100", "100.", "100.0", "100.00", and "100.000" are accepted as valid and equal.
- "100.01" and "100.99" are rejected, as the value cannot be stored accurately in an Integer64 primitive type.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

setByteOrderLocal

Signature  
setByteOrderLocal(architecture: Integer64): Integer64;

The setByteOrderLocal method of the Integer64 primitive type returns an integer that has the bytes ordered as required by the local node. The bytes of the receiver are assumed to be ordered as indicated by the architecture parameter.

The architecture parameter indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the getOSPlatform method of the Node class.

The architecture can be one of the Node class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>

Note  This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setByteOrderRemote

Signature  
setByteOrderRemote(architecture: Integer64): Integer64;

The setByteOrderRemote method of the Integer64 primitive type returns an integer that has the bytes ordered as required by the remote node indicated by the architecture parameter. The bytes of the receiver are assumed to be ordered as required by the local node.
The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the `getOSPlatform` method of the **Node** class.

The architecture can be one of the **Node** class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
<tr>
<td>No constant; that is, zero (0)</td>
<td>Reorders the bytes from network byte order (a standard for passing binary integers across a TCP/IP connection)</td>
</tr>
</tbody>
</table>

**Note** This method is not available on a Compact JADE node where it would result in a **1068 - Feature not available** exception.

### userCurrencyFormat

**Signature**

```java
userCurrencyFormat(fmt: CurrencyFormat): String;
```

The **userCurrencyFormat** method of the **Integer64** primitive type returns a string containing the receiver in the supplied currency format. To define your currency formats, use the Schema menu **Format** command from the Schema Browser.

**Notes** When you use a format in a JADE method, prefix your user currency format name with a dollar sign ($); for example, `userCurrencyFormat($MyCurrency)`.

You can use the **defineCurrencyFormat** method of the **CurrencyFormat** class if you want to create your own transient format objects and define a currency format that dynamically overrides the format for the locale at runtime. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

### userCurrencyFormatAndLcid

**Signature**

```java
userCurrencyFormatAndLcid(fmt: NumberFormat;
                           lcid: Integer): String;
```

The **userCurrencyFormatAndLcid** method of the **Integer64** primitive type returns a string containing the receiver in the currency format and locale specified in the fmt parameter and lcid parameter, respectively.

If the value of the fmt parameter is null, the settings for the locale specified in the lcid parameter are used. If the value of the lcid parameter is zero (0), the settings of the current locale are used.
If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**userNumberFormat**

**Signature**

```
userNumberFormat(fmt: NumberFormat): String;
```

The `userNumberFormat` method of the Integer64 primitive type returns a string containing the receiver in the supplied number format.

To define your numeric formats, use the Schema menu Format command from the Schema Browser.

**Notes**

When you use a format in a JADE method, prefix your user number format name with a dollar sign ($); for example, `userNumberFormat($MyNumber)`.

You can use the defineNumberFormat method from the NumberFormat class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at runtime. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**userNumberFormatAndLcid**

**Signature**

```
userNumberFormatAndLcid(fmt: NumberFormat;
                           lcid: Integer): String;
```

The `userNumberFormatAndLcid` method of the Integer64 primitive type returns a string containing the receiver in the number format and locale specified in the fmt parameter and lcid parameter, respectively.

If the value of the fmt parameter is null, the settings for the locale specified in the lcid parameter are used. If the value of the lcid parameter is zero (0), the settings of the current locale are used.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
MemoryAddress Type

A variable of type MemoryAddress is used to represent a memory address; that is, a void* pointer as used in C. The primary purpose for this primitive type is to interface with external C and C++ dynamic libraries being used as a parameter or a return type.

When an object with a MemoryAddress value is passed between nodes, it passes the value of the memory address. However, the memory address only has meaning for the node on which it was assigned a non-null value.

The following semantic rules apply to MemoryAddress values.

- A MemoryAddress variable can be assigned a null value but not the value zero (0).
- A MemoryAddress value can be compared to the null value or to another MemoryAddress value for equality or inequality.
- A MemoryAddress value can be assigned to another MemoryAddress variable.
- A MemoryAddress value cannot be changed by using an arithmetic operation.

**Note** Unlike other primitive types, a corresponding subclass of Array for MemoryAddress values does not exist in the RootSchema. If you require such an array, subclass the Array class in your user schema, selecting MemoryAddress as the membership.

For details about the methods defined in the MemoryAddress primitive type, see "MemoryAddress Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the JADE Developer’s Reference.

MemoryAddress Methods

The methods defined in the MemoryAddress primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjust</td>
<td>A changed memory address value</td>
</tr>
<tr>
<td>asBinary32</td>
<td>The value of the internal pointer as a 32-bit (4 byte) binary</td>
</tr>
<tr>
<td>asBinary64</td>
<td>The value of the internal pointer as a 64-bit (8 byte) binary</td>
</tr>
<tr>
<td>display</td>
<td>A string containing the receiver</td>
</tr>
<tr>
<td>isValid</td>
<td>true if the memory address is valid</td>
</tr>
</tbody>
</table>

**adjust**

**Signature** adjust(offset: Integer64): MemoryAddress;

The adjust method of the MemoryAddress primitive type adjusts the value of the receiver using the offset parameter and returns a new MemoryAddress value. If the MemoryAddress is not valid on the current node when this method is called, a 1443 (MemoryAddress is not valid for current Node) exception is raised.

The following code steps through a block of memory identified by iPtr in 1024 Byte blocks.

```java
vars
    iBytesLeft: Integer;
```
MemoryAddress Type

```plaintext
iBytesCopy: Integer;
iPtr: MemoryAddress;

begin
  ...
  while iBytesLeft > 0 do
    iBytesToCopy := iBytesLeft.min(1024);
    iPtr := iPtr.adjust(iBytesToCopy);
    iBytesLeft := iBytesLeft - iBytesToCopy;
  endwhile;
endwhile;
end;
```

**asBinary32**

**Signature**  
`asBinary32(): Binary;`

The **asBinary32** method of the **MemoryAddress** primitive type returns the value of the internal pointer as a 32-bit (4 byte) binary value.

An exception is raised if the process is not running in a 32-bit memory address space.

**asBinary64**

**Signature**  
`asBinary64(): Binary;`

The **asBinary64** method of the **MemoryAddress** primitive type returns the value of the internal pointer as a 64-bit (8 byte) binary value.

An exception is raised if the process is not running in a 64-bit memory address space.

**display**

**Signature**  
`display(): String;`

The **display** method of the **MemoryAddress** primitive type returns a string representing the value of the receiver.

**isValid**

**Signature**  
`isValid(): Boolean;`

The **isValid** method of the **MemoryAddress** primitive type returns **true** if the value of the receiver was assigned by a process on the current node; that is, the pointer is valid for the current operating system process.
Point Type

A variable of type **Point** is used to represent a point in two-dimensional space. A **Point** primitive type encapsulates two integer values: the \( x \) (horizontal) and \( y \) (vertical) coordinates.

You can use the **Point** primitive type to represent a position on the display or within a form or control. (When used to represent a position in a form or control, the integer values represent the units of the `scaleMode` property of the form or control.) Additionally, you can use the **Point** primitive type to represent any other two-dimensional data; for example, points on a graph.

For details about the methods defined in the **Point** primitive type, see "**Point Methods**", in the following subsection.

For details about converting primitive types, see "**Converting Primitive Types**", in Chapter 1 of the JADE Developer's Reference.

### Point Methods

The methods defined in the **Point** primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>display</code></td>
<td>Returns a string representing the value of the receiver</td>
</tr>
<tr>
<td><code>set</code></td>
<td>Sets the value of the receiver to a specified point</td>
</tr>
<tr>
<td><code>setX</code></td>
<td>Sets the ( x ) (horizontal) value of the receiver</td>
</tr>
<tr>
<td><code>setY</code></td>
<td>Sets the ( y ) (vertical) value of the receiver</td>
</tr>
<tr>
<td><code>x</code></td>
<td>Returns the ( x ) (horizontal) value of the receiver</td>
</tr>
<tr>
<td><code>y</code></td>
<td>Returns the ( y ) (vertical) value of the receiver</td>
</tr>
</tbody>
</table>

**display**

**Signature**

```java
display(): String;
```

The `display` method of the **Point** primitive type returns a string representing the value of the receiver.

**set**

**Signature**

```java
set(x, Integer;
   y: Integer) updating;
```

The `set` method of the **Point** primitive type sets the value of the receiver to a specified point.

The `x` and `y` parameters are integer values of the horizontal (x) and vertical (y) coordinates.

**setX**

**Signature**

```java
setX(x: Integer) updating;
```

The `setX` method of the **Point** primitive type sets the \( x \) (horizontal) value of the receiver.

The following examples show the use of the `setX` method.

```java
newPoint.setX(distance.Integer);
```
Encyclopaedia of Primitive Types

Point Type

Chapter 113

redrawGraph() updating;
vars
gLine : Object;
objectArray : ObjectArray;
begin
drawNew;
foreach gLine in objectArray do
gline.GLines.drawNew;
gline.GLines.lastPoint.setX(0);
gline.GLines.redrawGraph;
endforeach;
end;

setY

Signature setY(y: Integer) updating;
The setY method of the Point primitive type sets the y (vertical) value of the receiver.
The code fragments in the following examples show the use of the setY method.

gLine.lastPoint.setY(self.height.Integer);

newPoint.setY(yScale.heightOfYValue(dataArray.last).Integer);

x

Signature x(): Integer;
The x method of the Point primitive type returns the x (horizontal) value of the receiver.
The following example shows the use of the x method.

drawLine(lastPoint.x, lastPoint.y, distance + lastPoint.x,
newPoint.y, color);
lastPoint.setX(lastPoint.x + distance.Integer);
lastPoint.setY(newPoint.y);

y

Signature y(): Integer;
The y method of the Point primitive type returns the y (vertical) value of the receiver.
The following example shows the use of the y method.

newPoint.setY(yScale.verticalPixels.Integer - newPoint.y);
Real Type

Use the Real primitive type to represent a floating point number. A Real primitive type has a set of values that is a subset of real numbers. These values can be represented in floating point notation with a fixed number of digits.

Use a Real primitive type to store floating point numbers; for example, a temperature.

Real numbers are useful for computations involving very large or very small numbers, or when the range of magnitudes cannot be predicted.

Notes

Real primitive types provide fifteen digits of precision; that is, if you assign a Real value to a Real attribute or local variable and then retrieve the value in your method, only the first fifteen significant digits can be relied on for complete accuracy.

As a floating point number stores an approximation of the value that is accurate to fifteen significant digits, you should use an Integer, an Integer64, or a Decimal primitive type to store values where precision is required; for example, for monetary values.

Real numbers are stored internally using an eight-byte representation, providing fifteen significant digits of accuracy.

You can use the JadeEditMask class and TextBox class getTextAsReal and setTextFromReal methods to handle locale formatting for numeric fields.

For details about the constants and methods defined in the Real primitive type, see "Real Constants" and "Real Methods", in the following subsections. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the JADE Developer's Reference.

Real Constants

The constants provided by the Real primitive type are listed in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_Classification_NegInfinity</td>
<td>2</td>
</tr>
<tr>
<td>FP_Classification_Normal</td>
<td>6</td>
</tr>
<tr>
<td>FP_Classification_NotANumber</td>
<td>1</td>
</tr>
<tr>
<td>FP_Classification_PosInfinity</td>
<td>3</td>
</tr>
<tr>
<td>FP_Classification_SubNormal</td>
<td>5</td>
</tr>
<tr>
<td>FP_Classification_Zero</td>
<td>4</td>
</tr>
</tbody>
</table>

Real Methods

The methods defined in the Real primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>Returns the absolute value of the receiver</td>
</tr>
<tr>
<td>arccos</td>
<td>Returns the arc cosine of the receiver</td>
</tr>
<tr>
<td>arcsin</td>
<td>Returns the arc sine of the receiver</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>arctan</td>
<td>Returns the arc tangent of the receiver</td>
</tr>
<tr>
<td>arcTan2</td>
<td>Returns the arc tangent of a point (atan2 function)</td>
</tr>
<tr>
<td>cos</td>
<td>Returns the cosine of the receiver</td>
</tr>
<tr>
<td>currencyFormat</td>
<td>Returns a string in the currency format of the current locale</td>
</tr>
<tr>
<td>display</td>
<td>Returns the receiver as a string</td>
</tr>
<tr>
<td>exp</td>
<td>Returns the exponential $e$ to the power of the receiver</td>
</tr>
<tr>
<td>getFloatingPointClassification</td>
<td>Returns an integer indicating whether the receiver is a normal floating point value or a special value</td>
</tr>
<tr>
<td>infinity</td>
<td>Sets the receiver to the special \textit{positive infinity} value</td>
</tr>
<tr>
<td>isNaN</td>
<td>Returns \texttt{true} if the receiver has the special \textit{not a number} value</td>
</tr>
<tr>
<td>log</td>
<td>Returns the natural logarithm of the receiver</td>
</tr>
<tr>
<td>log10</td>
<td>Returns the base 10 logarithm of the receiver</td>
</tr>
<tr>
<td>max</td>
<td>Returns the larger value of the receiver and the specified \texttt{Real}</td>
</tr>
<tr>
<td>min</td>
<td>Returns the lesser value of the receiver and the specified \texttt{Real}</td>
</tr>
<tr>
<td>nan</td>
<td>Sets the receiver to the special \textit{not a number} value</td>
</tr>
<tr>
<td>numberFormat</td>
<td>Returns a string in the number format of the current locale</td>
</tr>
<tr>
<td>parseCurrencyWithCurrentLocale</td>
<td>Sets the receiver to the result of parsing a string representing a currency value for the current locale</td>
</tr>
<tr>
<td>parseCurrencyWithFmtAndLcid</td>
<td>Sets the receiver to the result of parsing a string representing a currency value for the specified format and the specified locale</td>
</tr>
<tr>
<td>parseNumberWithCurrentLocale</td>
<td>Sets the receiver to the result of parsing a string representing a number for the current locale</td>
</tr>
<tr>
<td>parseNumberWithFmtAndLcid</td>
<td>Sets the receiver to the result of parsing a string representing a number for the specified format and the specified locale</td>
</tr>
<tr>
<td>rounded</td>
<td>Returns an integer containing the rounded value of the receiver</td>
</tr>
<tr>
<td>rounded64</td>
<td>Returns a 64-bit integer containing the rounded value of the receiver</td>
</tr>
<tr>
<td>roundedTo</td>
<td>Returns the receiver rounded to the specified number of decimal places</td>
</tr>
<tr>
<td>roundedUp</td>
<td>Returns an integer containing the value of the receiver rounded up to the nearest whole number</td>
</tr>
<tr>
<td>roundedUp64</td>
<td>Returns a 64-bit integer containing the value of the receiver rounded up to the nearest whole number</td>
</tr>
<tr>
<td>setByteOrderLocal</td>
<td>Returns a \texttt{Real} that has the bytes ordered as required by the local node</td>
</tr>
<tr>
<td>setByteOrderRemote</td>
<td>Returns a \texttt{Real} that has the bytes ordered as required by the specified remote node</td>
</tr>
<tr>
<td>setFloatingPointClassification</td>
<td>Sets the receiver to a specified special value</td>
</tr>
<tr>
<td>sin</td>
<td>Returns the sine of the receiver</td>
</tr>
</tbody>
</table>
## Real Type

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt</td>
<td>Returns the square root of the receiver</td>
</tr>
<tr>
<td>tan</td>
<td>Returns the tangent of the receiver</td>
</tr>
<tr>
<td>truncated</td>
<td>Returns an integer containing the truncated value of the receiver</td>
</tr>
<tr>
<td>truncated64</td>
<td>Returns a 64-bit integer containing the truncated value of the receiver</td>
</tr>
<tr>
<td>truncatedTo</td>
<td>Returns the receiver truncated to the specified number of decimal places</td>
</tr>
<tr>
<td>userCurrencyFormat</td>
<td>Returns the receiver as a string in the specified currency format</td>
</tr>
<tr>
<td>userCurrencyFormatAndLcid</td>
<td>Returns the receiver as a string in the specified currency format for the specified locale</td>
</tr>
<tr>
<td>userNumberFormat</td>
<td>Returns the receiver as a string in the specified number format</td>
</tr>
<tr>
<td>userNumberFormatAndLcid</td>
<td>Returns the receiver as a string in the specified number format for the specified locale</td>
</tr>
</tbody>
</table>

### abs

**Signature**  
`abs() : Real;`

The `abs` method of the `Real` primitive type returns a real containing the absolute value of the receiver.

### arccos

**Signature**  
`arccos() : Real;`

The `arccos` method of the `Real` primitive type returns the arc cosine (or inverse cosine) of the receiver. An exception is raised if the receiver is invalid.

The resulting value represents an angle in degrees.

### arcsin

**Signature**  
`arcsin() : Real;`

The `arcsin` method of the `Real` primitive type returns the arc sine (or inverse sine) of the receiver. An exception is raised if the receiver is invalid.

The resulting value represents an angle in degrees.

### arctan

**Signature**  
`arctan() : Real;`

The `arctan` method of the `Real` primitive type returns the arc tangent (inverse tangent) of the receiver. An exception is raised if the receiver is invalid.

The resulting value represents an angle in degrees.
arcTan2

**Signature**  
`arcTan2(real: Real): Real;`

The `arcTan2` method of the `Real` primitive type returns the arc tangent (inverse tangent) of the `real` parameter divided by the receiver.

The method implements the `atan2` mathematical function relating to the angle subtended by a point in the Cartesian plane.

```
angle := self.arcTan2(real);
```

An exception is raised if the receiver is invalid. The resulting value represents an angle in degrees.

COS

**Signature**  
`cos(): Real;`

The `cos` method of the `Real` primitive type returns the cosine of the receiver. The receiver value represents an angle in degrees, and the resulting value is always in the range -1 through 1.

The following example shows the use of the `cos` method.

```
testReal();
vars
   realValue : Real;
begin
   realValue := 34;          // Defines the variable value
   write realValue.cos;     // Outputs 0.829037572555042
end;
```

currencyFormat

**Signature**  
`currencyFormat(): String;`

The `currencyFormat` method of the `Real` primitive type returns a string containing the receiver in the currency format defined for the current locale; for example, `$123.22` or `-123.225`. This can include currency symbols, thousands separators, sign characters, and decimal point characters.

The following examples show the use of the `currencyFormat` method.

```
tblPrices.text := company.currentPrice.currencyFormat;
testReal();
vars
   stringValue : String;
   realValue   : Real;
begin
   realValue := -123456.987;        // Defines the variable value
   write realValue;                  // Outputs -123456.987
   stringValue := realValue.currencyFormat; // Associates string and format
   write stringValue;                // Outputs ($123,456.99)
   realValue := 34.5;                // Outputs ($123,456.99)
   write realValue.max(40.6);        // Outputs 40.6
end;
```
Real Type

You can use the `defineCurrencyFormat` method of the `CurrencyFormat` class if you want to create your own transient format objects and define a currency format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

display

Signature `display(): String;`

The `display` method of the `Real` primitive type returns a string containing the receiver.

exp

Signature `exp(): Real;`

The `exp` method of the `Real` primitive type returns the exponential $e$ to the power of the receiver.

getFloatingPointClassification

Signature `getFloatingPointClassification(): Integer;`

The `getFloatingPointClassification` method of the `Real` primitive type returns an integer that indicates the whether the receiver is a normal floating point value or some other kind of special value.

Floating point numbers can have special values, such as infinity or NaN (Not a Number). Some floating point calculations may produce infinity or NaN as the result of an operation on invalid input operands.

The values returned by the `getFloatingPointClassification` method are represented by the constants defined for the `Real` primitive type and shown in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_Classification_NegInfinity</td>
<td>2</td>
</tr>
<tr>
<td>FP_Classification_Normal</td>
<td>6</td>
</tr>
<tr>
<td>FP_Classification_NotANumber</td>
<td>1</td>
</tr>
<tr>
<td>FP_Classification_PosInfinity</td>
<td>3</td>
</tr>
<tr>
<td>FP_Classification_SubNormal</td>
<td>5</td>
</tr>
<tr>
<td>FP_Classification_Zero</td>
<td>4</td>
</tr>
</tbody>
</table>

infinity

Signature `infinity(): Real updating;`

The `infinity` method of the `Real` primitive type sets the receiver to the special positive infinity value. It does this by calling the `setFloatingPointClassification` method with the `FP_Classification_PosInfinity` constant value as the parameter. The method also returns the positive infinity value.

See also the `isInfinity` method.
isInfinity

**Signature**

isInfinity(): Boolean;

The **isInfinity** method of the **Real** primitive type returns **true** if the value returned by the getFloatingPointClassification method is the constant **FP_Classification_PosInfinity**, which represents the special positive *infinity* value.

See also the **infinity** method.

isNaN

**Signature**

isNaN(): Boolean;

The **isNaN** method of the **Real** primitive type returns **true** if the value returned by the getFloatingPointClassification method is the constant **FP_Classification_NotANumber**, which represents the special *not a number* value.

See also the **nan** method.

log

**Signature**

log(): Real;

The **log** method of the **Real** primitive type returns the natural logarithm of the receiver. An exception is raised if the receiver is invalid.

log10

**Signature**

log10(): Real;

The **log10** method of the **Real** primitive type returns the base 10 logarithm of the receiver. An exception is raised if the receiver is invalid.

max

**Signature**

max(real: Real): Real;

The **max** method of the **Real** primitive type returns the larger value of the receiver and the real parameter.

If the value of the receiver is greater than the value of the real parameter, the value of the receiver is returned. If the value of the receiver is less than or equal to the value of the real parameter, the value of real is returned.

The following example shows the use of the **max** method.

```pascal
  testReal();
  vars
    realValue : Real;
  begin
    realValue := 34.5; // Defines the variable value
    write realValue.max(40.6); // Outputs 40.6
  end;
```
min

**Signature**  
\text{min}(real: Real): Real;

The \text{min} method of the \text{Real} primitive type returns the lesser value of the receiver and the \text{real} parameter.

If the value of the receiver is less than the value of the \text{real} parameter, the value of the receiver is returned. If the value of the receiver is greater than or equal to the value of the \text{real} parameter, the value of \text{real} is returned.

The following example shows the use of the \text{min} method.

```plaintext
testReal();
vars
    realValue : Real;
begin
    realValue := 34.5; // Defines the variable value
    write realValue.min(40.6); // Outputs 34.5
end;
```

nan

**Signature**  
\text{nan}(): Real updating;

The \text{nan} method of the \text{Real} primitive type sets the receiver to the special \text{not a number} value. It does this by calling the \text{setFloatingPointClassification} method with the \text{FP\_Classification\_NotANumber} constant value as the parameter. The method also returns the \text{not a number} value.

See also the \text{isNaN} method.

numberFormat

**Signature**  
\text{numberFormat}(): String;

The \text{numberFormat} method of the \text{Real} primitive type returns a string containing the receiver in the numeric format defined for the current locale; for example, \text{07456.357} or \text{7,456.38}. This can include thousands separators, sign characters, and decimal point characters.

You can use the \text{defineNumberFormat} method of the \text{NumberFormat} class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

If you do not define the \text{EnhancedLocaleSupport} parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to \text{false}, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The following example shows the use of the \text{numberFormat} method.

```plaintext
vars
    stringValue : String;
    realValue : Real;
begin
    realValue := -123456.987; // Defines the variable value
    write realValue; // Outputs -123456.987
    stringValue := realValue.numberFormat; // Associates string and format
```
The `parseCurrencyWithCurrentLocale` method of the `Real` primitive type parses the string specified in the `source` parameter to ensure that it matches the `Real` format of the current locale for currency character sequence, currency position, sign sequence, sign position, thousands separator, decimal point sequence, and character set. If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1860 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to zero (0).

This is equivalent to calling the `parseCurrencyWithFmtAndLcid` method, passing null in the `fmt` parameter and zero (0) in the `lcid` parameter.

If you do not define the `EnhancedLocaleSupport` parameter in the `[JadeEnvironment]` section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The `parseCurrencyWithFmtAndLcid` method of the `Real` primitive type parses the string specified in the `source` parameter using the specified format and locale, to ensure that it matches the format specified in the `fmt` parameter for currency character sequence, currency position, sign character sequence, sign position, thousands separator, decimal point sequence, and character set. If the value of the `fmt` parameter is null, the settings for the locale specified in the `lcid` parameter are used. If the value of the `lcid` parameter is zero (0), the settings of the current locale are used.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to zero (0).

The currency character sequence is optional but if it is included in the source, it must be correctly positioned as defined by the `NumberFormat` class `negativeFormat` property and the `CurrencyFormat` class `positiveFormat` property.

Thousands separator character sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

A space included in the sign and currency sequence is optional.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.
Only the first 15 significant digits of the value of the string are stored in the receiver. Any additional digits are rounded into the fifteenth digit and replaced with zeros. The number of decimal places before and after the decimal point is preserved. Leading zeros before the decimal point and trailing zeros after the decimal point are ignored. For example, "12345678901234567890" is parsed and results in 12345678901234600000 being stored in the receiver.

This method supports a maximum of 30 significant digits, whereas the fixed-point representation of a Real value can require up to 320 digits.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseNumberWithCurrentLocale**

**Signature**

```
parseNumberWithCurrentLocale(source: String; errOffset: Integer output): Integer updating;
```

The `parseNumberWithCurrentLocale` method of the `Real` primitive type parses the string specified in the `source` parameter to ensure that it matches the `Real` format of the current locale for sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to zero (0).

This is equivalent to calling the `parseNumberWithFmtAndLcid` method, passing null in the `fmt` parameter and zero (0) in the `lcid` parameter.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseNumberWithFmtAndLcid**

**Signature**

```
parseNumberWithFmtAndLcid(source: String; fmt: NumberFormat; lcid: Integer; errOffset: Integer output): Integer updating;
```

The `parseNumberWithFmtAndLcid` method of the `Real` primitive type parses the string specified in the `source` parameter using the specified format and locale, to ensure that it matches the format specified in the `fmt` parameter for sign sequence, sign position, thousands separator, decimal point sequence, and character set.

If the value of the `fmt` parameter is null, the settings for the locale specified in the `lcid` parameter are used. If the value of the `lcid` parameter is zero (0), the settings of the current locale are used.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to the invalid value.
The sign sequence is optional but if it is included in the source, it must be correctly positioned. A space included in the sign sequence is optional.

Thousands separator sequences are optional but if they are included in the source, each one must have at least one digit preceding and following it, and must occur before the decimal point (if any).

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

Only the first 15 significant digits of the value of the string are stored in the receiver. Any additional digits are rounded into the fifteenth digit and replaced with zeros.

The number of decimal places before and after the decimal point is preserved. Leading zeros before the decimal point and trailing zeros after the decimal point are ignored. For example, “12345678901234567890” is parsed and formatted results in 1234567890123460000 being stored in the receiver.

This method supports a maximum of 30 significant digits, whereas the fixed-point representation of a Real value can require up to 320 digits.

If native digits are allowed, if the first digit found in the source is a native digit, all subsequent digits must also be native. Similarly, if the first digit found is ASCII, all subsequent digits must also be ASCII.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

### rounded

**Signature**
```java
rounded(): Integer;
```

The **rounded** method of the **Real** primitive type returns an integer containing the rounded value of the receiver.

The following code fragments show the use of the **rounded** method.
```java
while count > 0 do
    tbl.rowHeight[count] := (tbl.height / tbl.rows).rounded;
    count := count - 1;
endwhile;

write (6.4).Real.rounded; // outputs 6 [0,1,2,3,4 are rounded down]
write (6.5).Real.rounded; // outputs 7 [5,6,7,8,9 are rounded up]

write (-6.4).Real.rounded; // outputs -6 [0,1,2,3,4 are rounded up]
write (-6.5).Real.rounded; // outputs -7 [5,6,7,8,9 are rounded down]
```

### rounded64

**Signature**
```java
rounded64(): Integer64;
```

The **rounded64** method of the **Real** primitive type returns a 64-bit integer containing the rounded value of the receiver.

The following examples show the use of the **rounded64** method.
```java
while count > 0 do
    tbl.rowHeight[count] := (tbl.height / tbl.rows).rounded64;
```
count := count - 1;
endwhile;

write (6.4).Real.rounded64; // outputs 6 [0,1,2,3,4 are rounded down]
write (6.5).Real.rounded64; // outputs 7 [5,6,7,8,9 are rounded up]
write (-6.4).Real.rounded64; // outputs -6 [0,1,2,3,4 are rounded up]
write (-6.5).Real.rounded64; // outputs -7 [5,6,7,8,9 are rounded down]

roundedTo

Signature roundedTo(decimalPlaces: Integer): Real;

The roundedTo method of the Real primitive type returns the receiver rounded to the number of decimal places specified in the decimalPlaces parameter.

The following example shows the use of the roundedTo method.

define vars
  realValue : Real;
define begin
  realValue := 340.5; // Defines the variable value
  realValue := (realValue / 27).roundedTo(2);
  write realValue; // Outputs 12.61
  realValue := 340.5; // Redefines the variable value
  realValue := (realValue / 27).roundedTo(5);
  write realValue; // Outputs 12.61111
end;

Note As Real values are implemented as floating point values, rounding may not return the expected value.

roundedUp

Signature roundedUp(): Integer;

The roundedUp method of the Real primitive type returns an integer containing the receiver rounded up to the nearest whole number.

The following examples show the use of the roundedUp method.

define integerValue := (columnWidth[column] / realValue).roundedUp;
define vars
  realValue : Real;
define begin
  realValue := 340.5; // Defines the variable value
  realValue := (realValue / 20).roundedUp;
  write realValue; // Outputs 18
end;

roundedUp64

Signature roundedUp64(): Integer64;

The roundedUp64 method of the Real primitive type returns a 64-bit integer containing the receiver rounded up to the nearest whole number.
The following examples show the use of the `roundedUp64` method.

```plaintext
integerValue := (columnWidth[column] / realValue).roundedUp64;
vars
    realValue : Real;
begin
    realValue := 340.5; // Defines the variable value
    realValue := (realValue / 20).roundedUp64;
    write realValue; // Outputs 18
end;
```

### setByteOrderLocal

**Signature**

`setByteOrderLocal(architecture: Integer): Real;`

The `setByteOrderLocal` method of the `Real` primitive type returns a real that has the bytes ordered as required by the local node. The bytes of the receiver are assumed to be ordered as indicated by the `architecture` parameter.

The `architecture` parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the `getOSPlatform` method of the `Node` class.

The architecture can be one of the `Node` class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>

**Note** This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

### setByteOrderRemote

**Signature**

`setByteOrderRemote(architecture: Integer): Real;`

The `setByteOrderRemote` method of the `Real` primitive type returns a real that has the bytes ordered as required by the remote node indicated by the `architecture` parameter. The bytes of the receiver are assumed to be ordered as required by the local node.

The `architecture` parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the `getOSPlatform` method of the `Node` class.
The architecture can be one of the Node class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>

**Note** This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

### setFloatingPointClassification

**Signature**

```plaintext
setFloatingPointClassification(classification: Integer) updating;
```

The `setFloatingPointClassification` method of the Real primitive type sets the receiver to a special value specified by the `classification` parameter.

The following table lists valid values for the `classification` parameter (for which you can use a constant defined for the Real primitive type) and the special value that results.

<table>
<thead>
<tr>
<th>classification Parameter</th>
<th>Resultant Special Value for Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_Classification_NegInfinity</td>
<td>Negative infinity.</td>
</tr>
<tr>
<td>FP_Classification_NotANumber</td>
<td>A Not a Number value. The result of an invalid operation; for example, attempting to find the square root of a negative number.</td>
</tr>
<tr>
<td>FP_Classification_PosInfinity</td>
<td>Positive infinity.</td>
</tr>
</tbody>
</table>

**Note** There are many possible Not a Number (NaN) representations. JADE returns a single NaN representation. A NaN does not compare equal to any floating-point number or NaN, even if the latter has an identical representation, as shown in the following code example.

```plaintext
vars
    real : Real;
begin
    real.setFloatingPointClassification(Real.FP_Classification_NotANumber);
    write real = real; // outputs false
end;
```

### sin

**Signature**

```plaintext
sin(): Real;
```

The `sin` method of the Real primitive type returns the sine of the receiver. The receiver value represents an angle in degrees, and the resulting value is always in the range -1 through 1.
The following example shows the use of the `sin` method.

```pascal
vars
  realValue : Real;
begin
  realValue := 340.5;  // Defines the variable value
  write realValue.sin;  // Outputs -0.333806859233771
end;
```

**sqrt**

**Signature**  `sqrt(): Real;`

The `sqrt` method of the `Real` primitive type returns the square root of the receiver.

The following example shows the use of the `sqrt` method.

```pascal
vars
  realValue : Real;
begin
  realValue := 340.5;  // Defines the variable value
  write realValue.sqrt;  // Outputs 18.4526420872459
end;
```

An exception is raised if the receiver is invalid.

**tan**

**Signature**  `tan(): Real;`

The `tan` method of the `Real` primitive type returns the tangent of the receiver. The receiver value represents an angle in degrees, and the resulting value is always in the range -1 through 1.

The following example shows the use of the `tan` method.

```pascal
vars
  realValue : Real;
begin
  realValue := 340.5;  // Defines the variable value
  write realValue.tan;  // Outputs -0.354118572530698
end;
```

**truncated**

**Signature**  `truncated(): Integer;`

The `truncated` method of the `Real` primitive type returns an integer containing the truncated value of the receiver.

The following example shows the use of the `truncated` method.

```pascal
vars
  realValue : Real;
begin
  realValue := 340.56789;  // Defines the variable value
  write realValue.truncated;  // Outputs 340
end;
```
truncated64

**Signature**  
truncated64(): Integer64;

The `truncated64` method of the `Real` primitive type returns a 64-bit integer containing the truncated value of the receiver.

The following example shows the use of the `truncated64` method.

```jade
vars
  realValue : Real;
begin
  realValue := 340.56789;  // Defines the variable value
  write realValue.truncated64;  // Outputs 340
end;
```

truncatedTo

**Signature**  
truncatedTo(decimalPlaces: Integer): Real;

The `truncatedTo` method of the `Real` primitive type returns the receiver truncated to the number of decimal places specified in the `decimalPlaces` parameter. The following example shows the use of the `truncatedTo` method.

```jade
vars
  realValue : Real;
begin
  realValue := 340.56789;  // Defines the variable value
  write realValue.truncatedTo(3);  // Outputs 340.567
end;
```

currencyFormat

**Signature**  
currencyFormat(fmt: CurrencyFormat): String;

The `currencyFormat` method of the `Real` primitive type returns a string containing the receiver in the currency format specified in the `fmt` parameter.

To define your currency formats, use the Schema menu `Format` command from the Schema Browser.

**Notes**  
When you use a format in a JADE method, prefix your user currency format name with a dollar sign ($); for example, `userCurrencyFormat($MyCurrency)`.

You can use the `defineCurrencyFormat` method of the `CurrencyFormat` class if you want to create your own transient format objects and define a currency format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
userCurrencyFormatAndLcid

Signature: userCurrencyFormatAndLcid(fmt: CurrencyFormat; lcid: Integer): String;

The userCurrencyFormatAndLcid method of the Real primitive type returns a string containing the receiver in the currency format and locale specified in the fmt parameter and lcid parameter, respectively.

If the value of the fmt parameter is null, the settings for the locale specified in the lcid parameter are used. If the value of the lcid parameter is zero (0), the settings of the current locale are used.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormat

Signature: userNumberFormat(fmt: NumberFormat): String;

The userNumberFormat method of the Real primitive type returns a string containing the receiver in the number format specified in the fmt parameter.

To define your numeric formats, use the Schema menu Format command from the Schema Browser.

Notes: When you use a format in a JADE method, prefix your user number format name with a dollar sign ($); for example, userNumberFormat($MyNumber).

You can use the defineNumberFormat method of the NumberFormat class if you want to create your own transient format objects and define a numeric format that dynamically overrides the format for the locale at runtime. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

userNumberFormatAndLcid

Signature: userNumberFormatAndLcid(fmt: NumberFormat; lcid: Integer): String;

The userNumberFormatAndLcid method of the Real primitive type returns a string containing the receiver in the number format and locale specified in the fmt parameter and lcid parameter, respectively.

If the value of the fmt parameter is null, the settings for the locale specified in the lcid parameter are used. If the value of the lcid parameter is zero (0), the settings of the current locale are used.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
**String Type**

Use the `String` primitive type to define `String` variables and attributes. A string contains zero or more characters. A `null` string is a string that has a zero length ("").

---

**Note** A string containing embedded `null` characters can be assigned to a local `String` variable and passed as a `String` parameter. Assigning a string containing embedded `null` characters to a `String` attribute may result in truncation of the string at the first `null` character.

To safely pass a string variable containing embedded `null` characters from JADE to an external method, define the string parameter in JADE as `io` usage.

---

When you specify a length less than 540 for a `String` attribute, it is embedded. Space is allocated within instances of the class to store a string with a length less than or equal to the specified length.

When you specify a length greater than or equal to 540 or you select the `Maximum Length` check box, which corresponds to 2,147,483,647 characters, for a `String` attribute, it is not embedded. It is stored in a separate variable-length object, a String Large Object (slob), which can store a string with a length less than or equal to the specified length. The amount of storage required for a slob is determined by the value of the string.

`String` variables can be bounded or unbounded, as shown in the following code fragment.

```java
vars
    str1 : String[100];  // Bounded - str1 can store a string with a
                        // length less than or equal to 100 characters
    str2 : String;       // Unbounded - str2 can store a string with a length
                        // less than or equal to 2,147,483,647 characters
begin
    str := "JADE Primitive Types";
    char := str[7];       // seventh character of the string, which is 'r'
```

The ordering relationship of the character values in corresponding positions sets the ordering between two string values. In strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a greater-than value; for example, Zs is greater than Z. Null strings can be equal only to other null strings.

To specify a substring `str[m:n]` of a string `str`, two integers separated by a colon (:) character are used. In substrings, the first integer is the start position, and the second integer (following the colon (:)) character is the length of the substring or `end`, to indicate the end of the string. The first character is defined as being at position 1.

If the length of a substring is zero (0), a null string ("") is returned.

A variable of type `Character` can be used to reference a single character in a string, in effect treating the string as an array of characters, as shown in the following code fragment.

For details about the methods defined in the `String` primitive type, see "String Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the JADE Developer’s Reference.
## String Methods

The methods defined in the `String` primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asANSI</td>
<td>Returns ANSI String values as Binary values in a Unicode environment</td>
</tr>
<tr>
<td>asDate</td>
<td>Returns a date based on the contents from the receiving string</td>
</tr>
<tr>
<td>asGuid</td>
<td>Returns a binary representation based on the class identifier (clsid) of the receiving string</td>
</tr>
<tr>
<td>asObject</td>
<td>Returns an object reference based on the contents of the oid-like receiving string and an optional lifetime indication</td>
</tr>
<tr>
<td>asOid</td>
<td>Returns an object reference based on the contents of the receiving string</td>
</tr>
<tr>
<td>asStringUtf8</td>
<td>Returns a locale-sensitive conversion of the receiving string in UTF8 format</td>
</tr>
<tr>
<td>asUuid</td>
<td>Returns a binary by formatting the string as a Universally Unique Identifier (UUID)</td>
</tr>
<tr>
<td>base64Decode</td>
<td>Returns a binary value resulting from decoding a Base64-encoded message</td>
</tr>
<tr>
<td>bufferAddress</td>
<td>Returns the value of the pointer to the internal buffer as an integer</td>
</tr>
<tr>
<td>bufferMemoryAddress</td>
<td>Returns the value of the pointer to the internal buffer as a memory address</td>
</tr>
<tr>
<td>compareEql</td>
<td>Returns <code>true</code> if the receiver is equal to a specified string</td>
</tr>
<tr>
<td>compareGeneric</td>
<td>Returns an integer showing if the receiver is greater than, equal to, or less than a specified character</td>
</tr>
<tr>
<td>compareGeq</td>
<td>Returns <code>true</code> if the receiver is greater than or equal to a specified string</td>
</tr>
<tr>
<td>compareGtr</td>
<td>Returns <code>true</code> if the receiver is greater than a specified string</td>
</tr>
<tr>
<td>compareLeq</td>
<td>Returns <code>true</code> if the receiver is less than or equal to a specified string</td>
</tr>
<tr>
<td>compareLss</td>
<td>Returns <code>true</code> if the receiver is less than the value of a specified string</td>
</tr>
<tr>
<td>compareNeq</td>
<td>Returns <code>true</code> if the receiver is not equal to a specified string</td>
</tr>
<tr>
<td>compressToBinary</td>
<td>Returns a compressed binary representation of the receiver</td>
</tr>
<tr>
<td>display</td>
<td>Returns a string containing the receiver</td>
</tr>
<tr>
<td>fillString</td>
<td>Fills the receiving string with the specified string</td>
</tr>
<tr>
<td>firstCharToLower</td>
<td>Converts an uppercase first character in the receiving string to lowercase</td>
</tr>
<tr>
<td>firstCharToUpper</td>
<td>Converts a lowercase first character in the receiving string to uppercase</td>
</tr>
<tr>
<td>getHugeTokens</td>
<td>Returns an array of the tokens not greater than 2047 characters in the receiver</td>
</tr>
<tr>
<td>getNextToken</td>
<td>Returns the next token in the receiver</td>
</tr>
<tr>
<td>getTokens</td>
<td>Returns an array of the tokens not greater than 62 characters in the receiver</td>
</tr>
<tr>
<td>isByte</td>
<td>Returns <code>true</code> if the receiver is a string representation of a valid byte value</td>
</tr>
<tr>
<td>isDecimal</td>
<td>Returns <code>true</code> if the receiver is a string representation of a valid decimal value</td>
</tr>
<tr>
<td>isInteger</td>
<td>Returns <code>true</code> if the receiver is a string representation of a valid integer value</td>
</tr>
<tr>
<td>isInteger64</td>
<td>Returns <code>true</code> if the receiver is a string representation of a valid 64-bit integer value</td>
</tr>
</tbody>
</table>
Method | Description
---|---
isReal | Returns true if the receiver is a string representation of a valid real number
length | Returns the current length of a string variable or attribute
makeString | Returns a string of the specified length filled with the value of the receiver
makeXMLCDATA | Returns a new string of the receiver prepended with <![CDATA[ and appended with ]]> 
maxLength | Returns the declared maximum length of a string variable
padBlanks | Returns a copy of the receiving string padded to the specified length with trailing blanks (spaces)
padLeadingZeros | Returns a copy of the receiving string padded to the specified length with leading zeros
plainTextToStringUtf8 | Returns a UTF8 string with escaped character sequences replaced by UTF8 characters
pos | Returns an integer containing the position of a substring in the receiver
replaceChar | Replaces all occurrences of a character with another character
reverse | Returns a string containing the reversed characters in the receiving string
reversePos | Returns the position of the last occurrence of a substring in the receiving string
reversePosIndex | Returns the position of the last occurrence of a substring within a substring of the receiving string
scanUntil | Returns a substring of the receiving string starting from the specified index up to (but not including) the first occurrence of any of the specified characters
scanWhile | Returns a substring of the receiving string starting from the specified index up to (but not including) the first occurrence of any character other than the specified characters
toLower | Returns a copy of the receiving string with all uppercase characters converted to lowercase
toUpper | Returns a copy of the receiving string with all lowercase characters converted to uppercase
trimBlanks | Returns a copy of the receiving string with blanks (spaces) trimmed from both ends of the receiver
trimLeft | Returns a copy of the receiving string with the leading blanks (spaces) removed
trimRight | Returns a copy of the receiving string with trailing blanks (spaces) trimmed from the end of the receiver

**asANSI**

**Signature**  
asANSI(lcid: Integer): Binary;

The asANSI method of the String primitive type returns the receiving string converted to a Binary value using the character set of the code page for the locale specified by the lcid parameter. You can use this method in a Unicode environment to produce ANSI strings in a binary format.

An exception is raised on the first source character that cannot be represented in the code page of the specified locale; for example, a multi-byte Chinese character encountered when the locale is specified as New Zealand.
asDate

Signature    asDate(): Date;

The asDate method of the String primitive type returns a date based on the contents of the receiving string. If the receiving string does not contain a valid date, "invalid" is returned. The data value must represent one of the following date formats.

- `dd-MMM-yy` (for example, 30-Aug-11)
- `dd/MM/yy` (for example, 30/08/11)
- `MMM dd, yy` (for example, Aug 30, 11)
- `yyyy:MM:dd` (for example, 2011:08:30)

Any non-alphanumeric character can be used as a delimiter.

JADE converts a two-digit year as follows.

- If the current year is equal to or less than 50, all dates default to the current century.
- If the current year is greater than 50, dates that have a year greater than 50 default to the current century.
- If the current year is greater than 50, dates equal to or less than 50 default to the next century.

**Note** You should always use four-digit years in your applications.

The following example shows the use of the asDate method.

```javascript
vars
dateValue : Date;
begin
dateValue := "15 May 2010".asDate;  // 15 May 2010
dateValue := "15-May-2010".asDate;  // 15 May 2010
dateValue := "15/5/2010".asDate;  // 15 May 2010
dateValue := "May 15, 2010".asDate;  // 15 May 2010
dateValue := "2010:5:15".asDate;  // 15 May 2010
dateValue := "29/2/2011".asDate;  // "*invalid*"
end;
```

asGuid

Signature    asGuid(): Binary;

The asGuid method of the String primitive type returns the class identifier (clsid) receiver string as a Globally Unique Identifier (GUID) binary representation. Binary GUID representations of string class identifiers (used in ActiveX control and automation libraries, for example) take less space than a visual string representation. This method raises an exception if the receiver is not a GUID string in the following format.

"{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx}"

See also the Binary primitive type asGuidString method.
**asObject**

**Signature**

```
asObject(): Object;
```

The **asObject** method of the **String** primitive type returns an object reference based on the contents of the oid-like receiving string based on class numbers, followed by an optional lifetime indication.

This method is the inverse of the **Object** class **getObjectStringForObject** method.

The form of the oid-like string can be one of the following.

- `class-number.instd_id`
- `class-number.instd_id.parent-class-number`
- `class-number.instd_id.parent-class-number.subLevel.subId`

The optional lifetime can be `'(t)'`, to indicate a transient object, or `'(s)'`, to indicate a shared transient object. If the optional lifetime is absent, it indicates a persistent object.

The following code fragments are examples of the use of the **asObject** method.

```plaintext
// return persistent instance of class number 16401
obj := '16401.1'.asObject;
// return transient instance of class number 16401
obj := '16401.1 (t)'.asObject;
// return shared transient instance of class number 16401
obj := '16401.1 (s)'.asObject;
```

**Tip** The **asObject** method is useful for debugging from a Workspace method to inspect a specific oid; for example, an oid returned in an exception dialog.

For details about returning an object reference based on the contents of the receiving string, see the **String** primitive type **asOid** method.

**asOid**

**Signature**

```
asOid(): Object;
```

The **asOid** method of the **String** primitive type returns an object reference based on the contents of the receiving string.

This method is the inverse of the **Object** class **getOidStringForObject** method.

The following example shows the use of the **asOid** method.

```plaintext
begin
  // inspect an instance of an object, in this case 2048.5
  '2048.5'.asOid.inspect;
end;
```

For details about returning an object reference based on the contents of the oid-like receiving string based on class numbers and a following optional lifetime indication, see the **String** primitive type **asObject** method.

**asStringUtf8**

**Signature**

```
asStringUtf8(lcid: Integer): StringUtf8;
```

The **asStringUtf8** method of the **String** primitive type returns the receiving string converted to a UTF8 string value.
In an ANSI environment, the conversion uses the character set of the code page for the locale specified by the lcid parameter.

**asUuid**

**Signature** asUuid(): Binary;

The **asUuid** method of the **String** primitive type returns a binary by formatting the string as a Universally Unique Identifier (UUID).

If the string is not formatted as a valid UUID representation (that is, as returned by the **Binary** primitive type **uuidAsString** method), exception 1407 (*Invalid argument passed to method*) is raised.

The code fragment in the following example shows the use of the **asUuid** method.

```plaintext
vars
    str : String;
    bin : Binary;
begin
    str := "4dfc912a-b466-01d0-1027-000085823b00";
    bin := str.asUuid();
```

**base64Decode**

**Signature** base64Decode(): Binary;

The **base64Decode** method of the **String** primitive type returns a **Binary** value resulting from the decoding of a Base64-encoded message. A Base64-encoded message contains characters from the following alphabet.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
```

The message may also contain line-break characters (Cr and Lf) inserted by the Base64 encoding algorithm; these are ignored by the decoder, as are any characters that are not in the Base64 alphabet.

Use the **base64Encode** or **base64EncodeNoCrLf** method on the **String** primitive type to encode a binary value using Base64 encoding.

The following example shows the use of the **base64Decode** method.

```plaintext
vars
    bin: Binary;
    file: File;
begin
    create file;
    file.fileName := "d:\temp\harry.jpg";
    file.kind := File.Kind_Binary;
    file.open;
    bin := file.readBinary(file.fileLength);
    write 'original length = ' & bin.length.String;
    write 'base64Encode length = ' & bin.base64Encode().length.String;
    write 'base64EncodeNoCrLf length = ' &
       bin.base64EncodeNoCrLf().length.String;
    write 'base64Decode length = ' &
       bin.base64Decode().base64Decode().length.String;
    file.close;
```
bufferAddress

**Signature**  
bufferAddress(): Integer;

The bufferAddress method of the String primitive type returns an integer containing the value of the pointer to the internal buffer that contains the string. This value may be required when a JADE Binary type value is being mapped to a structured record type for a call to an external function. Call the bufferAddress method to determine the address of the buffer when an external function requires a data structure to contain a pointer to a second structure.

The use of the bufferAddress method for the String primitive type is similar to that for the Binary primitive type. For an example of using the bufferAddress method of the Binary primitive type to initialize the Windows SECURITY_DESCRIPTOR and SECURITY_ATTRIBUTES structures, see bufferAddress, under "Binary Type".

The code fragment in the following example shows the use of the bufferAddress method when copying clipboard data directly into a JADE string.

```plaintext
call copyString(str.bufferAddress, locked);
call globalUnlock(locked);
```

**Caution**  
Do not use this method to pass the address of a string to an external function that will be executed by a presentation client. If an external function is called from an application server method and executed by a different process (the presentation client), the memory address is not valid and will almost certainly result in a jade.exe (thin client) fault in the called function.

bufferMemoryAddress

**Signature**  
bufferMemoryAddress(): MemoryAddress;

The bufferMemoryAddress method of the String primitive type returns a memory address containing the value of the pointer to the internal buffer that contains the string. This value may be required when a JADE String type value is being mapped to a structured record type for a call to an external function.

Call the bufferMemoryAddress method to determine the address of the buffer when an external function requires a data structure to contain a pointer to a second structure.

The use of the bufferMemoryAddress method for the String primitive type is similar to that for the Binary primitive type.

For an example of using the bufferMemoryAddress method of the Binary primitive type to initialize the Windows SECURITY_DESCRIPTOR and SECURITY_ATTRIBUTES structures, see bufferMemoryAddress, under "Binary Type".

The code fragment in the following example shows the use of the bufferMemoryAddress method when copying clipboard data directly into a JADE string.

```plaintext
call copyString(str.bufferMemoryAddress, locked);
call globalUnlock(locked);
```
**Caution**  Do not use this method to pass the address of a string to an external function that will be executed by a presentation client. If an external function is called from an application server method and executed by a different process (the presentation client), the memory address is not valid and will almost certainly result in a `jade.exe` (thin client) fault in the called function.

**compareEql**

**Signature**  
```java
compareEql(rhs: String;
  bIgnoreCase: Boolean;
  bUseLocale: Boolean;
  locale: Locale): Boolean;
```

The `compareEql` method of the `String` primitive type returns `true` if the receiver is equal to the value of the `rhs` parameter; otherwise, it returns `false`.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note**  The relational binary comparison operator `(?=)`, documented in Chapter 1 of the JADE Developer’s Reference, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is `false`:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also `false`.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is `true`:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.
  ```java
  recv.compareEql(lhs, true, false, null);
  recv.toLowerCase = lhs.toLowerCase;
  ```
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the `compareEql` method.

```java
write "Alice".compareEql("alice", true, false, null);  // Outputs true
write "Alice".compareEql("alice", false, false, null);  // Outputs false
```
compareGeneric

**Signature**

```java
compareGeneric(rhs: String;
    bIgnoreCase: Boolean;
    bUseLocale: Boolean;
    locale: Locale): Integer;
```

The `compareGeneric` method of the `String` primitive type compares the receiver with the value of the `rhs` parameter and returns one of the following values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Returned if the receiver is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative integer</td>
<td>Less than the right-hand side value represented by the <code>rhs</code> parameter</td>
</tr>
<tr>
<td>Zero (0)</td>
<td>Equal to the right-hand side value represented by the <code>rhs</code> parameter</td>
</tr>
<tr>
<td>Positive integer</td>
<td>Greater than the right-hand side value represented by the <code>rhs</code> parameter</td>
</tr>
</tbody>
</table>

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operators (`<`, `<=`, `=`, `>=`, `>`, `<>`), documented in Chapter 1 of the JADE Developer’s Reference, use a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is **false**:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment:

  ```java
  recv.compareGeneric(lhs, true, false, null);
  (recv.toLower>lhs.toLower).Integer - (recv.toLower<lhs.toLower).Integer;
  ```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the `compareGeneric` method.

```java
vars
locale : Locale;
begin
    write "alice".compareGeneric("carol", false, false, null); // Outputs -1
    write "bob".compareGeneric("bob", false, false, null); // Outputs 0
    write "carol".compareGeneric("alice", false, false, null); // Outputs 1
```

EnydoPrim - 7.1.08
// Comparisons with accented characters using binary and locale sort orders
locale := currentSchema.getLocale("5129");
write "àcute".compareGeneric("zebra", false, false, null); // Outputs 1
write "àcute".compareGeneric("zebra", false, true, locale); // Outputs -1

compareGeq

**Signature**

```java
compareGeq(rhs: String; bIgnoreCase: Boolean; bUseLocale: Boolean; locale: Locale): Boolean;
```

The `compareGeq` method of the `String` primitive type returns `true` if the receiver is greater than or equal to the value of the `rhs` parameter; otherwise, it returns `false`.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note**

The relational binary comparison operator (`>=`), documented in Chapter 1 of the JADE Developer’s Reference, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is `false`:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also `false`.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is `true`:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.
  ```java
  recv.compareGeq(lhs, true, false, null);
  recv.toLowerCase >= lhs.toLowerCase;
  ```
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the `compareGeq` method.

```java
vars
locale : Locale;
begin
write "alice".compareEq("carol", false, false, null); // Outputs false
write "bob".compareEq("bob", false, false, null); // Outputs true
write "carol".compareEq("alice", false, false, null); // Outputs true
// Comparisons with accented characters using binary and locale sort orders
locale := currentSchema.getLocale("5129");
```
The `compareGtr` method of the `String` primitive type returns `true` if the receiver is greater than the value of the `rhs` parameter; otherwise, it returns `false`.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

Note: The relational binary comparison operator (>) documented in Chapter 1 of the JADE Developer’s Reference, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is `false`:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also `false`.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is `true`:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

```java
recv.compareGtr(lhs, true, false, null);
recv.toLower > lhs.toLower;
```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the `compareGtr` method.

```java
vars
locale : Locale;
begin
write "alice".compareGtr("carol", false, false, null); // Outputs false
write "bob".compareGtr("bob", false, false, null); // Outputs false
write "carol".compareGtr("alice", false, false, null); // Outputs true
// Comparisons with accented characters using binary and locale sort orders
locale := currentSchema.getLocale("5129");
write "acute".compareGtr("zebra", false, false, null); // Outputs true
write "acute".compareGtr("zebra", false, true, locale); // Outputs false
```
**compareLeq**

**Signature**

```java
compareLeq(rhs: String;
   bIgnoreCase: Boolean;
   bUseLocale: Boolean;
   locale: Locale): Boolean;
```

The `compareLeq` method of the `String` primitive type returns `true` if the receiver is less than or equal to the value of the `rhs` parameter; otherwise, it returns `false`.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note**  The relational binary comparison operator (`<=`), documented in Chapter 1 of the *JADE Developer’s Reference*, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is `false`:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also `false`.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is `true`:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

  ```java
  recv.compareLeq(lhs, true, false, null);
  recv.toLowerCase <= lhs.toLowerCase;
  ```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the `compareLeq` method.

```java
vars
   locale : Locale;
begin
   write "alice".compareLeq("carol", false, false, null); // Outputs true
   write "bob".compareLeq("bob", false, false, null); // Outputs true
   write "carol".compareLeq("alice", false, false, null); // Outputs false
   // Comparisons with accented characters using binary and locale sort orders
   locale := currentSchema.getLocale("5129");
   write "àcute".compareLeq("zebra", false, false, null); // Outputs false
   write "àcute".compareLeq("zebra", false, true, locale); // Outputs true
```
**compareLss**

**Signature**
```java
compareLss(rhs: String;
bIgnoreCase: Boolean;
bUseLocale: Boolean;
locale: Locale): Boolean;
```

The **compareLss** method of the **String** primitive type returns **true** if the receiver is less than the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operator (<), documented in Chapter 1 of the **JADE Developer’s Reference**, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:
- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:
- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.
  ```java
  recv.compareLss(lhs, true, false, null);
  recv.toLowerCase < lhs.toLowerCase;
  ```
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

The code fragment in the following example shows the use of the **compareLss** method.
```
vars
  locale : Locale;
begin
  write "alice".compareLss("carol", false, false, null); // Outputs true
  write "bob".compareLss("bob", false, false, null); // Outputs false
  write "carol".compareLss("alice", false, false, null); // Outputs false
  // Comparisons with accented characters using binary and locale sort orders
  locale := currentSchema.getLocale("5129");
  write "àcute".compareLss("zebra", false, false, null); // Outputs false
  write "àcute".compareLss("zebra", false, true, locale); // Outputs true
```
**compareNeq**

**Signature**

```java
compareNeq(rhs: String;
bIgnoreCase: Boolean;
bUseLocale: Boolean;
locale: Locale): Boolean;
```

The `compareNeq` method of the `String` primitive type returns `true` if the receiver is not equal to the value of the `rhs` parameter; otherwise, it returns `false`.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operator (`<>`), documented in Chapter 1 of the JADE Developer's Reference, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is `false`:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also `false`.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is `true`:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

  ```java
  recv.compareNeq(lhs, true, false, null);
  recv.toLower <> lhs.toLower;
  ```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

The code fragment in the following example shows the use of the `compareNeq` method.

```java
write "Alice".compareNeq("alice", true, false, null);  // Outputs false
write "Alice".compareNeq("alice", false, false, null);  // Outputs true
```
**compressToBinary**

**Signature**  
compressToBinary(typeAndOption: Integer): Binary;

The `compressToBinary` method of the `String` primitive type returns a compressed binary representation of the string of the receiver using the ZLIB compression value specified in the `typeAndOption` parameter, one of the `Binary` type constants listed in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression_ZLib</td>
<td>1402</td>
<td>String and binary compression to binary using ZLIB level 5 (256^5 + 122)</td>
</tr>
<tr>
<td>Compression_ZLibFast</td>
<td>378</td>
<td>String and binary compression to binary using ZLIB level 1 (256*1 + 122)</td>
</tr>
<tr>
<td>Compression_ZLibSmall</td>
<td>2426</td>
<td>String and binary compression to binary using ZLIB level 9 (256^9 + 122)</td>
</tr>
</tbody>
</table>

**Notes**  
This method adds the type byte to the front of the compressed binary. This type byte is ignored when the value is used in a JADE system but if the data is to be passed to an external library, it is your responsibility to remove the type byte, if necessary.

You cannot concatenate the results of multiple `compressToBinary` method calls.

You must use the `Binary` primitive type `uncompressToQString` method to uncompress a binary value from this binary representation.

**display**

**Signature**  
display(): String;

The `display` method of the `String` primitive type returns a string enclosed in double quotation marks (""") containing the receiver.

If the length of the receiver is zero (0), the string "<null>" is returned.

**fillString**

**Signature**  
fillString(string: String) updating;

The `fillString` method of the `String` primitive type fills the receiving string with repeated copies of the string specified in the `string` parameter up to the length of the receiver.

The following example shows the use of the `fillString` method.

```plaintext
vars
    stringValue : String;
begin
    stringValue := 'hello world';
    stringValue.fillString('foo');
    write stringValue;    // Outputs 'foofoofoofo'
end;
```
firstCharToLower

Signature  

The `firstCharToLower` method of the `String` primitive type converts an uppercase first character in the receiving string to lowercase, according to the conventions of the current locale.

The following example shows the use of the `firstCharToLower` method.

```pascal
vars
  stringValue : String;
begin
  stringValue := 'HELLO WORLD';
  stringValue.firstCharToLower;
  write stringValue;  // Outputs 'hELLO WORLD'
end;
```

firstCharToUpper

Signature  

The `firstCharToUpper` method of the `String` primitive type converts a lowercase first character in the receiving string to uppercase, according to the conventions of the current locale.

The following example shows the use of the `firstCharToUpper` method.

```pascal
vars
  stringValue : String;
begin
  stringValue := 'hello world';
  stringValue.firstCharToUpper;
  write stringValue;  // Outputs 'Hello world'
end;
```

getHugeTokens

Signature  

The `getHugeTokens` method of the `String` primitive type returns an array of the tokens in the receiver that have a length in the range 0 through 2047 characters. For details about tokens, see the `getNextToken` method.

The following example shows the use of the `getHugeTokens` method.

```pascal
vars
  stringValue : String;
  hugeStringArray : HugeStringArray;
begin
  stringValue := 'this:is/a:string';
  hugeStringArray := stringValue.getHugeTokens;
  write hugeStringArray [1];  // Outputs this
  write hugeStringArray [2];  // Outputs is
  write hugeStringArray [3];  // Outputs a
  write hugeStringArray [4];  // Outputs string
end;
```
String Type

getNextToken

**Signature**

```
getNextToken(int: Integer io): String;
```

The `getNextToken` method of the `String` primitive type returns the next token in the receiver; that is, it returns the string from the current value of the `int` parameter to the next delimiter.

**Tip**

As the `String:scanUntil` method (which supersedes this `getNextToken` method) provides increased functionality and flexibility, you may want to use that method instead.

The string delimiter can be any of the following characters.

- Colon character (`:`)
- Semicolon character (`;`)
- Stroke character (`/`)
- Double quotation character (`"`)
- Single quotation character (`'`)
- Space
- Tab
- End of string

To define the position from which the next delimiter is returned, specify the starting position in the `int` parameter in a method.

The following example shows the use of the `getNextToken` method.

```
vars
  str : String;
  token : Integer;
begin
  str := 'this is a string';
  token := 1;
  write str.getNextToken(token) & ' ' & token.String; // Outputs 'this 6'
  write str.getNextToken(token) & ' ' & token.String; // Outputs 'is 9'
  write str getNextToken(token) & ' ' & token.String; // Outputs 'a 11'
  write str getNextToken(token) & ' ' & token.String; // Outputs 'string 17'
end;
```

The `getNextToken` method returns `null` when the end of string is reached.

getTokens

**Signature**

```
getTokens(): StringArray;
```

The `getTokens` method of the `String` primitive type returns an array of the tokens in the receiver that have a length not greater than 62 characters. For details about tokens, see the `getNextToken` method.

The following example shows the use of the `getTokens` method.

```
vars
  stringValue : String;
```
String Type

```pascal
stringArray : StringArray;
begin
        stringValue := 'this:is/a;string';
        stringArray := stringValue.getTokens;
        write stringArray [1];    // Outputs this
        write stringArray [2];    // Outputs is
        write stringArray [3];    // Outputs a
        write stringArray [4];    // Outputs string
end;
```

**isByte**

**Signature**
isByte(): Boolean;

The *isByte* method of the **String** primitive type returns **true** if the receiver represents a valid byte value; that is, in the range zero (0) through **255**; otherwise, it returns **false**.

The following example shows the use of the *isByte* method.

```pascal
vars
        stringValue : String;
begin
        stringValue := '+123';
        write stringValue.isByte; // Outputs true
        stringValue := '+321';
        write stringValue.isByte; // Outputs false
end;
```

**isDecimal**

**Signature**
isDecimal(): Boolean;

The *isDecimal* method of the **String** primitive type returns **true** if the receiver represents a valid decimal value; otherwise, it returns **false**.

The following example shows the use of the *isDecimal* method.

```pascal
vars
        stringValue : String;
begin
        stringValue := '+123.456';
        write stringValue.isDecimal; // Outputs true
        stringValue := '+123,456';
        write stringValue.isDecimal; // Outputs false
end;
```

**isInteger**

**Signature**
isInteger(): Boolean;

The *isInteger* method of the **String** primitive type returns **true** if the receiver represents an integer value; otherwise, it returns **false**.
The following example shows the use of the `isInteger` method.

```pascal
vars
  stringValue : String;
begin
  stringValue := '+123';
  write stringValue.isInteger;  // Outputs true
  stringValue := '+123.456';
  write stringValue.isInteger;  // Outputs false
end;
```

**isInteger64**

**Signature**  
`isInteger64(): Boolean;`

The `isInteger64` method of the `String` primitive type returns `true` if the receiver represents a 64-bit integer value; otherwise, it returns `false`.

The following example shows the use of the `isInteger64` method.

```pascal
vars
  stringValue : String;
begin
  stringValue := '+123';
  write stringValue.isInteger64;  // Outputs true
  stringValue := '+123.456';
  write stringValue.isInteger64;  // Outputs false
end;
```

**isReal**

**Signature**  
`isReal(): Boolean;`

The `isReal` method of the `String` primitive type returns `true` if the receiver represents a valid real value; otherwise, it returns `false`.

The following example shows the use of the `isReal` method.

```pascal
vars
  stringValue : String;
begin
  stringValue := '+123.456';
  write stringValue.isReal;  // Outputs true
  stringValue := '+123,456';
  write stringValue.isReal;  // Outputs false
end;
```

**length**

**Signature**  
`length(): Integer;`

The `length` method of the `String` primitive type returns the actual length of the value that has been assigned to an embedded `String` property; for example, if you declared a `String` property with length of 30 but the value stored is of length 20, the `length` method returns 20.
The following example shows the use of the `length` method.

```pascal
vars
  stringValue : String;
begin
  stringValue := 'hello world';
  write stringValue.length;  // Outputs 11
end;
```

**makeString**

**Signature**

```pascal
makeString(length: Integer): String;
```

The `makeString` method of the `String` primitive type returns a string of the length specified in the `length` parameter filled with the value of the receiver. If the receiver is null (""), the returned string is filled with spaces.

If the value of the `length` parameter is less than or equal to zero (0), an empty string is returned.

The following example shows the use of the `makeString` method.

```pascal
vars
  strValue : String;
begin
  strValue := "*";
  write strValue.makeString(10);  // Outputs **********
  strValue := "*--*--*--";
  write strValue.makeString(10);  // Outputs *--*--*--*
  strValue := null;
  write strValue.makeString(10);  // Outputs (ten spaces)
end;
```

**makeXMLCData**

**Signature**

```pascal
makeXMLCData(): String;
```

The `makeXMLCData` method of the `String` primitive type returns a new string of the receiver prepended with `<![CDATA[ and appended with ]]>`. Note that the receiver is not modified in any way.

The following example is a receiver string.

```xml
<greeting>Hello, world!</greeting>
```

The returned string of this receiver string is as follows.

```xml
<![CDATA[<greeting>Hello, world!</greeting>]]>
```

Call this method for any string for which you do not want the framework to interpret the XML special characters (that is, `<`, `>`, `&`, and `"`).

**maxLength**

**Signature**

```pascal
maxLength(): Integer;
```

The `maxLength` method of the `String` primitive type returns the declared maximum length of a string variable. If the string variable maximum length has not been declared, the value of the `Max_UnboundedLength` global constant in the `SystemLimits` category is returned.
The following example shows the use of the `maxLength` method.

```plaintext
vars
    stringValue : String[100];
begin
    stringValue := 'hello world';
    write stringValue.maxLength;  // Outputs 100
end;
```

**padBlanks**

**Signature**   padBlanks(int: Integer): String;

The `padBlanks` method of the `String` primitive type returns a string of the length specified in the `int` parameter, consisting of the receiving string padded with appended (trailing) spaces.

If the string is longer than the integer value, it is not truncated but the whole string is returned.

The following example shows the use of the `padBlanks` method.

```plaintext
vars
    stringValue : String;
begin
    stringValue := 'Alfonso:';
    write stringValue.padBlanks(10) & '123 Sesame St.';
    // Outputs 'Alfonso: 123 Sesame St.'
end;
```

**padLeadingZeros**

**Signature**   padLeadingZeros(int: Integer): String;

The `padLeadingZeros` method of the `String` primitive type returns a string of the length specified in the `int` parameter, consisting of the receiving string padded with leading zeros.

The following example shows the use of the `padLeadingZeros` method.

```plaintext
vars
    stringValue : String;
begin
    stringValue := '123.45';
    write stringValue.padLeadingZeros(10);
    // Outputs '0000123.45'
end;
```

**plainTextToStringUtf8**

**Signature**   plainTextToStringUtf8(utf8: StringUtf8 output): Integer;

The `plainTextToStringUtf8` method of the `String` primitive type assigns a UTF8 string to the value of the `utf8` output parameter. The plain text of the receiver is converted to UTF8 format with any escaped character sequences being replaced by the appropriate UTF8 character.

The method returns zero (0) if the entire string is converted successfully. If an invalid escaped character sequence is encountered, the `plainTextToStringUtf8` method returns the offset of the first character in error and the `utf8` parameter contains the result of the conversion up to the invalid character.
In the following example, the character sequence \&copy; is recognized as a valid character © but the character sequence \&cool; is not recognized. The invalid character starts at position 14.

vars
  str : String;
  str8 : StringUtf8;
begin
  str := "\&copy; Jade \&cool; Software";
  write str.plainTextToStringUtf8(str8);  // 14
  write str8;                           // © Jade
end;

pos

Signature pos(substr: String; 
  start: Integer): Integer;

The pos method of the String primitive type returns an integer containing the position of a substring in a string. The substring is specified by the substr parameter. The search for the substring begins at the position specified by the start parameter.

The start parameter must be greater than zero (0) and less than or equal to the length of the receiver. If the substr or the start parameter is greater than the length of the receiver, this method returns zero (0). This method returns zero (0) if the specified substring is not found.

Note The character search is case-sensitive.

The following example shows the use of the pos method.

vars
  stringValue : String;
begin
  stringValue := 'position example';
  write stringValue.pos('pos', 1);   // Outputs 1
  write stringValue.pos('pos', 10);  // Outputs 0
end;

replaceChar

Signature replaceChar(char: Character; 
   withChar: Character): updating;

The replaceChar method of the String primitive type replaces all occurrences of the character specified in the char parameter with the character specified in the withChar parameter.

Note The character replacement is case-sensitive.

The following example shows the use of the replaceChar method.

vars
  stringValue : String;
begin
  stringValue := "zhis example shows character replacement";
  write stringValue; // Outputs: zhis example shows character replacement
  stringValue.replaceChar("z", "T");
String Type  

write stringValue;  // Outputs: This example shows character replacement
end;

reverse

Signature reverse(): String;

The reverse method of the String primitive type returns a string consisting of the receiving string with the position of all characters reversed. For example, a string that contains "abcde" is returned as "edcba". The following example shows the use of the reverse method.

vars
  stringValue : String;
begin
  stringValue := 'abcde';
  write stringValue.reverse;  // Outputs 'edcba'
end;

reversePos

Signature reversePos(substr: String): Integer;

The reversePos method of the String primitive type returns the position of the last occurrence of the substring specified in the substr parameter in the receiving string.

Note  The character search is case-sensitive.

The following example shows the use of the reversePos method.

vars
  stringValue : String;
begin
  stringValue := "Reverse position example";
  write stringValue.reversePos('pos');  // Outputs 9
end;

reversePosIndex

Signature reversePosIndex(substr: String;
index: Integer): Integer;

The reversePosIndex method of the String primitive type returns the position of the last occurrence of the substring specified in the substr parameter, in a string formed from the first character of the receiving string up to (and including) the character position specified in the index parameter.

Notes  The character search is case-sensitive.

The value of the index parameter cannot exceed the length of the receiving string.

The following example shows the use of the reversePosIndex method.

vars
  stringValue : String;
  count : Integer;
begin
Encyclopaedia of Primitive Types

String Type

```
stringValue := "car->taxi->bus->train";
count := stringValue.length;
while count > 0 do
    count := stringValue.reversePosIndex("->", count);
    write count;
    count := count - 1;
endwhile;
// Outputs 15
// Outputs 10
// Outputs 4
// Outputs 0
end;
```

This method returns zero (0) if the specified substring is not found.

**scanUntil**

**Signature**

```
scanUntil(delimiters: String;
          index: Integer io): String;
```

The `scanUntil` method of the **String** primitive type returns a substring of the receiving string starting from the index specified in the `index` parameter up to (but not including) the first occurrence of any of the characters specified in the `delimiters` parameter.

The index of the delimiting character is returned in the second parameter.

If a delimiting character is not found, the return value is the remainder of the receiving string (from the specified index) and an index value of zero (0) is returned in the second parameter.

**Note**  The character search is case-sensitive.

The following example shows the use of the `scanUntil` method.

```
vars
    stringValue : String;
    pos          : Integer;
begin
    stringValue := "this:is/a;string";
    pos := 1;
    write stringValue.scanUntil("/:;", pos);  // Outputs this
    pos := pos+1;
    write stringValue.scanUntil("/:;", pos);  // Outputs is
end;
```

See also the **String** primitive type **getNextToken** and **scanWhile** methods.

**scanWhile**

**Signature**

```
scanWhile(characters: String;
          index:   Integer io): String;
```

The `scanWhile` method of the **String** primitive type returns a substring of the receiving string starting from the index specified in the `index` parameter up to (but not including) the first occurrence of any character other than the characters specified in the `characters` parameter.

The index of the delimiting character is specified in the second parameter.
If a delimiting character is not found, the return value is the remainder of the string and an index value of zero (0) is returned in the second parameter, as shown in the following example.

```pascal
vars
  i: Integer;
  s: String;
begin
  i := 3;
  s := '0246'.scanWhile('0123456789', i);
  write '<' & s & ' ' & i.String;  // outputs <46> 0, not <> 0
end;
```

**Notes** The character search is case-sensitive.

The delimiting character is any character that is not specified in the `characters` parameter.

The following example shows the use of the `scanWhile` method.

```pascal
vars
  stringValue : String;
  pos      : Integer;
begin
  stringValue := "this:is/a;string";
  pos := 1;
  write stringValue.scanWhile("abcdefghijklmnopqrstuvwxyz", pos);
  // Outputs this
  pos := pos+1;
  write stringValue.scanWhile("abcdefghijklmnopqrstuvwxyz", pos);
  // Outputs is
end;
```

See also the `String` primitive type `scanUntil` method.

**toLower**

**Signature** `toLower(): String;`

The `toLower` method of the `String` primitive type returns a copy of the receiving string with all uppercase characters converted to lowercase, according to the conventions of the current locale.

The following example shows the use of the `toLower` method.

```pascal
vars
  stringValue : String;
begin
  stringValue := "UPPERCASE TEXT CAN LOOK THREATENING"
  write stringValue.toLower;
  // Outputs uppercase text can look threatening
end;
```

**toUpper**

**Signature** `toUpper(): String;`

The `toUpper` method of the `String` primitive type returns a copy of the receiving string with all lowercase characters converted to uppercase, according to the conventions of the current locale.
The following example shows the use of the `toUpperCase` method.

```pascal
vars
    stringValue : String;
begin
    stringValue := "lowercase";
    write stringValue.toUpperCase;  // Outputs LOWERCASE
end;
```

### trimBlanks

**Signature**    trimBlanks(): String;

The `trimBlanks` method of the `String` primitive type returns a copy of the receiving string with blanks (spaces) trimmed from both ends of the receiver.

The following example shows the use of the `trimBlanks` method.

```pascal
vars
    stringValue : String;
begin
    stringValue := 'some text'.trimBlanks;
    write stringValue;  // Outputs 'some text'
end;
```

### trimLeft

**Signature**    trimLeft(): String;

The `trimLeft` method of the `String` primitive type returns a copy of the receiving string with leading blanks (spaces) removed.

The following example shows the use of the `trimLeft` method.

```pascal
vars
    stringValue : String;
begin
    stringValue := 'some text'.trimLeft;
    write stringValue;  // Outputs 'some text '
end;
```

### trimRight

**Signature**    trimRight(): String;

The `trimRight` method of the `String` primitive type returns a copy of the receiving string with trailing blanks (spaces) trimmed from the end of the receiver.

The following example shows the use of the `trimRight` method.

```pascal
vars
    stringValue : String;
begin
    stringValue := 'some text '.trimRight;
    write stringValue;  // Outputs ' some text'
end;
```
StringUtf8 Type

Use the StringUtf8 primitive type to define StringUtf8 variables and attributes; that is, strings that have been encoded in the UTF8 format. This allows all valid Unicode characters to be used even in an ANSI system. A character string contains zero or more characters. A null string is a string that has a zero length (""'). You can access characters in a string as components of an array.

When you specify a length less than 540 for a StringUtf8 attribute, it is embedded. Space is allocated within instances of the class to store a string with a length less than or equal to the specified length.

When you specify a length greater than or equal to 540 or you select the Maximum Length check box (which corresponds to 2,147,483,647 characters) for a StringUtf8 attribute, it is not embedded. It is stored in a separate variable-length object, a String Large Object (slob), which can store a string with a length less than or equal to the specified length. The amount of storage required for a slob is determined by the value of the string.

StringUtf8 variables can be bounded or unbounded, as shown in the following code fragment.

```java
vars
    s1 : StringUtf8[100]; // Bounded - s1 can store a string with a length less than or equal to 100 characters
    s2 : StringUtf8; // Unbounded - s2 can store a string with a length less than or equal to 2,147,483,647 characters
```

The ordering relationship of the character values in corresponding positions sets the ordering between two string values. In strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a greater-than value; for example, Z is greater than Z. Null strings can be equal only to other null strings.

To specify a substring str[m:n] of a string str, two integers separated by a colon (:) character are used. The first integer indicates the start position and the second integer is the length of the substring. In place of the second integer, end indicates the substring extends to the end of the string. For a substring starting at the first character of the string, the first integer would be 1.

If the length of a substring is zero (0), a null string (""') is returned.

**Note** You can ignore the fact that a non-ASCII character in a UTF8 string requires more than one byte of storage, as the start position and length integers are based on character positions rather than on byte positions.

A StringUtf8 literal is enclosed in double ("'') or single ("') quotation marks, and is usually preceded by an at sign (@), as shown in the following example.

```javascript
stringUtf8 := @"Jade Software";
```

If all the characters are US-ASCII characters, as in the preceding example, the @ sign is optional.

The StringUtf8 literal can contain a non-US-ASCII character, by enclosing a value representing the character between an ampersand (&) character and a semicolon (;) character, as shown in the following examples.

```javascript
stringUtf8 := @"Copyright &copy; Jade Software";
stringUtf8 := @"Copyright &amp;169; Jade Software";
stringUtf8 := @"Copyright &amp;xA9; Jade Software";
```

In the first example, a character entity reference as defined in the HTML 4 standard is used. In the second and third examples, the value of the Unicode code point of the character in decimal and in hexadecimal is used.
A variable of type StringUtf8 can be used to reference a single character in a string, in effect treating the string as an array of one-character UTF8 strings, as shown in the following code fragment.

```pascal
vars
    str1 : StringUtf8;
    str2 : StringUtf8;
begin
    str1 := @"JADE Primitive Types";
    str2 := str1[7];  // UTF8 string consisting of seventh character 'r'
```

For details about the methods defined in the StringUtf8 primitive type, see "StringUtf8 Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the JADE Developer's Reference.

### StringUtf8 Methods

The methods defined in the StringUtf8 primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asANSI</td>
<td>Returns ANSI String values as Binary values in a Unicode environment</td>
</tr>
<tr>
<td>asDate</td>
<td>Returns a date based on the contents of the receiving UTF8 string</td>
</tr>
<tr>
<td>asPlainText</td>
<td>Returns a string with a character entity escape sequence replacing each non-US-ASCII character</td>
</tr>
<tr>
<td>asString</td>
<td>Returns multiple-byte or code page-sensitive values as String values in an ANSI environment</td>
</tr>
<tr>
<td>bufferMemoryAddress</td>
<td>Returns the value of the pointer to the internal buffer as a memory address</td>
</tr>
<tr>
<td>byteOffsetFromCharacterIndex</td>
<td>Returns the byte offset for a specified index of a character within the receiving UTF8 string</td>
</tr>
<tr>
<td>characterIndexFromByteOffset</td>
<td>Returns the index of a character for a specified byte offset within the receiving UTF8 string</td>
</tr>
<tr>
<td>compareEqI</td>
<td>Returns true if the receiver is equal to a specified UTF8 string</td>
</tr>
<tr>
<td>compareGeneric</td>
<td>Returns an integer showing if the receiver is greater than, equal to, or less than a UTF8 string</td>
</tr>
<tr>
<td>compareGeq</td>
<td>Returns true if the receiver is greater than or equal to a specified UTF8 string</td>
</tr>
<tr>
<td>compareGtr</td>
<td>Returns true if the receiver is greater than a specified UTF8 string</td>
</tr>
<tr>
<td>compareLeq</td>
<td>Returns true if the receiver is less than or equal to a specified UTF8 string</td>
</tr>
<tr>
<td>compareLss</td>
<td>Returns true if the receiver is less than the value of a specified UTF8 string</td>
</tr>
<tr>
<td>compareNeq</td>
<td>Returns true if the receiver is not equal to a specified UTF8 string</td>
</tr>
<tr>
<td>compressToBinary</td>
<td>Returns a compressed binary representation of a UTF8 string</td>
</tr>
<tr>
<td>display</td>
<td>Returns the receiver encoded in the character set of the default code page for the current locale</td>
</tr>
<tr>
<td>firstCharToLower</td>
<td>Converts an uppercase first character in the receiving UTF8 string to lowercase</td>
</tr>
<tr>
<td>firstCharToUpper</td>
<td>Converts a lowercase first character in the receiving UTF8 string to uppercase</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>isValid</td>
<td>Returns <code>true</code> if the receiver represents a valid UTF8 string</td>
</tr>
<tr>
<td>length</td>
<td>Returns the current number of characters in the receiver</td>
</tr>
<tr>
<td>maxLength</td>
<td>Returns the declared maximum number of characters in the receiver</td>
</tr>
<tr>
<td>padBlanks</td>
<td>Returns a copy of the receiving string padded to the specified length with trailing blanks (spaces)</td>
</tr>
<tr>
<td>padLeadingZeros</td>
<td>Returns a copy of the receiving string padded to the specified length with leading zeros</td>
</tr>
<tr>
<td>pos</td>
<td>Returns an integer containing the character index of the start of a UTF8 substring in the receiver</td>
</tr>
<tr>
<td>posUsingByteOffset</td>
<td>Returns an integer containing the byte offset of the start of a UTF8 substring in the receiver</td>
</tr>
<tr>
<td>replaceChar</td>
<td>Replaces all occurrences of a character with another character</td>
</tr>
<tr>
<td>reverse</td>
<td>Returns a UTF8 string containing the reversed characters in the receiving UTF8 string</td>
</tr>
<tr>
<td>reversePos</td>
<td>Returns the position of the last occurrence of a substring in the receiving UTF8 string</td>
</tr>
<tr>
<td>reversePosIndex</td>
<td>Returns the position of the last occurrence of a substring within a substring of the UTF8 string</td>
</tr>
<tr>
<td>scanUntil</td>
<td>Returns a substring of the receiving UTF8 string starting from the specified index up to (but not including) the first occurrence of any of the specified characters</td>
</tr>
<tr>
<td>scanWhile</td>
<td>Returns a substring of the receiving UTF8 string starting from the specified index up to (but not including) the first occurrence of any character other than the specified characters</td>
</tr>
<tr>
<td>size</td>
<td>Returns the number of bytes required to store the receiver excluding the end-of-string character</td>
</tr>
<tr>
<td>substringAtByteOffset</td>
<td>Returns a UTF8 substring with a specified length starting with the character at a specified byte offset</td>
</tr>
<tr>
<td>toLower</td>
<td>Returns a copy of the receiving UTF8 string with all uppercase characters converted to lowercase</td>
</tr>
<tr>
<td>toUpper</td>
<td>Returns a copy of the receiving UTF8 string with all lowercase characters converted to uppercase</td>
</tr>
<tr>
<td>trimBlanks</td>
<td>Returns a copy of the receiving UTF8 string with blanks (spaces) trimmed from both ends of the receiver</td>
</tr>
<tr>
<td>trimLeft</td>
<td>Returns a copy of the receiving UTF8 string with the leading blanks (spaces) removed</td>
</tr>
<tr>
<td>trimRight</td>
<td>Returns a copy of the receiving UTF8 string with trailing blanks (spaces) trimmed from the end of receiver</td>
</tr>
</tbody>
</table>
asANSI

Signature    asANSI(lcid: Integer): Binary;

The asANSI method of the StringUtf8 primitive type returns the receiving UTF8 string converted to a Binary value using the character set of the code page for the locale specified by the lcid parameter.

You can use this method in a Unicode environment to produce ANSI strings in a binary format.

asDate

Signature    asDate(): Date;

The asDate method of the StringUtf8 primitive type returns a date based on the contents of the receiving string.

If the receiving string does not contain a valid date, "invalid" is returned.

The data value must represent one of the following date formats.

- *dd-MMM-yy* (for example, 30-Aug-11)
- *dd/MM/yy* (for example, 30/08/11)
- *MMM dd, yy* (for example, Aug 30, 11)
- *yyyy:MM:dd* (for example, 2011:08:30)

Any non-alphanumeric character can be used as a delimiter.

JADE converts a two-digit year as follows.

- If the current year is equal to or less than 50, all dates default to the current century.
- If the current year is greater than 50, dates that have a year greater than 50 default to the current century.
- If the current year is greater than 50, dates equal to or less than 50 default to the next century.

**Note** Always use four-digit years in your applications.

The following example shows the use of the asDate method.

```pascal
vars
dateValue : Date;
begin
  dateValue := @"15 May 2010".asDate; // 15 May 2010
  dateValue := @"15-May-2010".asDate; // 15 May 2010
  dateValue := @"15/5/2010".asDate; // 15 May 2010
  dateValue := @"May 15, 2010".asDate; // 15 May 2010
  dateValue := @"2010:5:15".asDate; // 15 May 2010
  dateValue := @"29/2/2011".asDate; // "*invalid*"
end;
```
asPlainText

Signature asPlainText(): String;

The asPlainText method of the StringUtf8 primitive type returns a string containing the US-ASCII characters from the receiving UTF8 string with the non-US-ASCII characters replaced with a character entity escape sequence using an entity name if possible; otherwise a hexadecimal value.

ASCII control characters (excluding carriage returns, line feeds, and tabs) are converted to hexadecimal escape sequences. The ampersand and semicolon characters are converted to \& and \; respectively.

The following code example shows the difference between using the asPlainText method and converting to a String value.

```pascal
vars
    str8: StringUtf8;
begin
    str8 := "Copyright &copy;";
    write str8.asPlainText; // outputs "Copyright &copy;&semi;"
    write str8.String;     // outputs "Copyright ©;"
end;
```

asString

Signature asString(lcid: Integer): String;

The asString method of the StringUtf8 primitive type returns the receiving UTF8 string converted to a String value.

In an ANSI environment, the conversion uses the character set of the code page for the locale specified by the lcid parameter.

bufferMemoryAddress

Signature bufferMemoryAddress(): MemoryAddress;

The bufferMemoryAddress method of the StringUtf8 primitive type returns a memory address containing the value of the pointer to the internal buffer that contains the UTF8 string. This value may be required when a JADE StringUtf8 type value is being mapped to a structured record type for a call to an external function. Call the bufferMemoryAddress method to determine the address of the buffer when an external function requires a data structure to contain a pointer to a second structure.

The use of the bufferMemoryAddress method for the StringUtf8 primitive type is similar to that for the Binary primitive type.

For an example of using the bufferMemoryAddress method of the Binary primitive type to initialize the Windows SECURITY_DESCRIPTOR and SECURITY_ATTRIBUTES structures, see bufferMemoryAddress, under "Binary Type".

The code fragment in the following example shows the use of the bufferMemoryAddress method when copying clipboard data directly into a JADE string.

```pascal
call copyString(stringUtf8.bufferMemoryAddress, locked);
call globalUnlock(locked);
```
**Caution**  Do not use this method to pass the address of a UTF8 string to an external function that will be executed by a presentation client. If an external function is called from an application server method and executed by a different process (the presentation client), the memory address is not valid and will almost certainly result in a **jade.exe** (thin client) fault in the called function.

### byteOffsetFromCharacterIndex

**Signature**  
```plaintext
byteOffsetFromCharacterIndex(index: Integer): Integer;
```

The `byteOffsetFromCharacterIndex` method of the `StringUtf8` primitive type returns the byte offset for the character specified by the `index` parameter within the receiving UTF8 string.

In the following code example, the first character of the UTF8 string `str8` requires two bytes with the remaining four characters requiring one byte each. The second character therefore starts at offset three (3).

```plaintext
vars
    str8 : StringUtf8;
begin
    str8 := @"©2007";
    write str8.byteOffsetFromCharacterIndex(1); // writes 1
    write str8.byteOffsetFromCharacterIndex(2); // writes 3
    write str8.byteOffsetFromCharacterIndex(3); // writes 4
    write str8.byteOffsetFromCharacterIndex(4); // writes 5
    write str8.byteOffsetFromCharacterIndex(5); // writes 6
end;
```

### characterIndexFromByteOffset

**Signature**  
```plaintext
characterIndexFromByteOffset(offset: Integer): Integer;
```

The `characterIndexFromByteOffset` method of the `StringUtf8` primitive type returns the index of the character that starts at the byte offset specified in the `offset` parameter within the receiving UTF8 string, or after that offset, that is, the method scans from the offset position forwards to find the next character. If there is no next character, an exception is raised.

In the following code example, the two characters of the string `str8` require two bytes and three bytes in the UTF8 encoding. The first character starts at offset one (1) and the second character at offset three (3).

```plaintext
vars
    str8: StringUtf8;
begin
    str8 := @"©€";
    write str8.characterIndexFromByteOffset(1); // writes 1
    write str8.characterIndexFromByteOffset(2); // writes 2
    write str8.characterIndexFromByteOffset(3); // writes 2
    write str8.characterIndexFromByteOffset(4); // raises 1413 exception
    write str8.characterIndexFromByteOffset(5); // raises 1413 exception
end;
```
**compareEql**

**Signature**

```java
compareEql(rhs: StringUtf8;
  bIgnoreCase: Boolean;
  bUseLocale: Boolean;
  locale: Locale): Boolean;
```

The `compareEql` method of the `StringUtf8` primitive type returns `true` if the receiver is equal to the value of the `rhs` parameter; otherwise, it returns `false`.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operator (≠), documented in Chapter 1 of the JADE Developer's Reference, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is `false`:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also `false`.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-sensitive comparison using the sort order of the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is `true`:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed; for example, the first of the following code fragments is equivalent to the second code fragment.

  ```java
  recv.compareEql(lhs, true, false, null);
  ```

  ```java
  recv.toLowerCase = lhs.toLowerCase;
  ```

- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is `true` and the value of the `locale` parameter is not null.

**compareGeneric**

**Signature**

```java
compareGeneric(rhs: StringUtf8;
  bIgnoreCase: Boolean;
  bUseLocale: Boolean;
  locale: Locale): Integer;
```

The `compareGeneric` method of the `StringUtf8` primitive type compares the receiver with the value of the `rhs` parameter and returns one of the following values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Returned if the receiver is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative integer</td>
<td>Less than the right-hand side value represented by the <code>rhs</code> parameter</td>
</tr>
<tr>
<td>Zero (0)</td>
<td>Equal to the right-hand side value represented by the <code>rhs</code> parameter</td>
</tr>
<tr>
<td>Positive integer</td>
<td>Greater than the right-hand side value represented by the <code>rhs</code> parameter</td>
</tr>
</tbody>
</table>
Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operators (<, <=, =, >=, >, <>), documented in Chapter 1 of the JADE Developer's Reference, use a strict binary value comparison.

If the value of the `bgnoreCase` parameter is false:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also false.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is true and the value of the `locale` parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is true and the value of the `locale` parameter is not null.

If the value of the `bgnoreCase` parameter is true:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed.
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is true and the value of the `locale` parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is true and the value of the `locale` parameter is not null.

**compareGeq**

**Signature**

```java
compareGeq(rhs: StringUtf8;
            bgnoreCase: Boolean;
            bUseLocale: Boolean;
            locale: Locale): Boolean;
```

The `compareGeq` method of the `StringUtf8` primitive type returns true if the receiver is greater than or equal to the value of the `rhs` parameter; otherwise, it returns false.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operator (>=), documented in Chapter 1 of the JADE Developer's Reference, uses a strict binary value comparison.

If the value of the `bgnoreCase` parameter is false:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also false.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is true and the value of the `locale` parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is true and the value of the `locale` parameter is not null.
If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed.
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

**compareGtr**

**Signature**  
```java
compareGtr(rhs: StringUtf8;  
    bIgnoreCase: Boolean;  
    bUseLocale: Boolean;  
    locale: Locale): Boolean;
```

The **compareGtr** method of the **StringUtf8** primitive type returns **true** if the receiver is greater than the value of the **rhs** parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note**  
The relational binary comparison operator (>): documented in Chapter 1 of the JADE Developer’s Reference, uses a strict binary value comparison.

If the value of the **bIgnoreCase** parameter is **false**:

- A strict binary value comparison is performed if the value of the **bUseLocale** parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

If the value of the **bIgnoreCase** parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed.
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the **bUseLocale** parameter is **true** and the value of the **locale** parameter is not null.

**compareLeq**

**Signature**  
```java
compareLeq(rhs: StringUtf8;  
    bIgnoreCase: Boolean;  
    bUseLocale: Boolean;  
    locale: Locale): Boolean;
```

The **compareLeq** method of the **StringUtf8** primitive type returns **true** if the receiver is less than or equal to the value of the **rhs** parameter; otherwise, it returns **false**.
Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operator (\(\leq\)), documented in Chapter 1 of the *JADE Developer’s Reference*, uses a strict binary value comparison.

If the value of the `bIgnoredCase` parameter is false:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also false.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is true and the value of the `locale` parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is true and the value of the `locale` parameter is not null.

If the value of the `bIgnoredCase` parameter is true:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed.
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is true and the value of the `locale` parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is true and the value of the `locale` parameter is not null.

**compareLss**

**Signature**

```java
compareLss(rhs: StringUtf8;
  bIgnoredCase: Boolean;
  bUseLocale: Boolean;
  locale: Locale): Boolean;
```

The `compareLss` method of the `StringUtf8` primitive type returns `true` if the receiver is less than the value of the `rhs` parameter; otherwise, it returns `false`.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

**Note** The relational binary comparison operator (<), documented in Chapter 1 of the *JADE Developer’s Reference*, uses a strict binary value comparison.

If the value of the `bIgnoredCase` parameter is false:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also false.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is true and the value of the `locale` parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is true and the value of the `locale` parameter is not null.
If the value of the `bIgnoreCase` parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed.
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is not null.

**compareNeq**

**Signature**  
```java  
compareNeq(StringUtf8 rhs;  
bIgnoreCase: Boolean;  
bUseLocale: Boolean;  
locale: Locale): Boolean;  
```  

The `compareNeq` method of the `StringUtf8` primitive type returns **true** if the receiver is not equal to the value of the `rhs` parameter; otherwise, it returns **false**.

Parameters enable you to make the comparison case-sensitive or case-insensitive, and to use the sort order associated with a locale or the strict binary sort order. (These are the same comparison options that you can specify on dictionary keys.)

---

**Note**  
The relational binary comparison operator (\(<\>\)), documented in Chapter 1 of the *JADE Developer's Reference*, uses a strict binary value comparison.

If the value of the `bIgnoreCase` parameter is **false**:

- A strict binary value comparison is performed if the value of the `bUseLocale` parameter is also **false**.
- A case-sensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is null.
- A case-sensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is not null.

If the value of the `bIgnoreCase` parameter is **true**:

- A case-insensitive binary value comparison for characters less than Decimal 254 is performed.
- A case-insensitive comparison using the sort order of the current locale of the process is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is null.
- A case-insensitive comparison using the specified locale is performed if the value of the `bUseLocale` parameter is **true** and the value of the `locale` parameter is not null.
**compressToBinary**

**Signature**  
`compressToBinary(typeAndOption: Integer): Binary;`

The `compressToBinary` method of the `StringUtf8` primitive type returns a compressed binary representation of the UTF8 string of the receiver using the ZLIB compression routine specified in the `typeAndOption` parameter, using one of the `Binary` type constants listed in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression_ZLib</td>
<td>1402</td>
<td>String and binary compression to binary using ZLIB level 5 (256*5 + 122)</td>
</tr>
<tr>
<td>Compression_ZLibFast</td>
<td>378</td>
<td>String and binary compression to binary using ZLIB level 1 (256*1 + 122)</td>
</tr>
<tr>
<td>Compression_ZLibSmall</td>
<td>2426</td>
<td>String and binary compression to binary using ZLIB level 9 (256*9 + 122)</td>
</tr>
</tbody>
</table>

**Notes**  
This method adds the type byte to the front of the compressed binary. This type byte is ignored when the value is used in a JADE system but if the data is to be passed to an external library, it is your responsibility to remove the type byte, if necessary.

You cannot concatenate the results of multiple `compressToBinary` method calls.

You must use the `Binary` type `uncompressToStringUtf8` method to uncompress a binary value from this binary representation.

**display**

**Signature**  
`display(): String;`

The `display` method of the `StringUtf8` primitive type returns a string enclosed in double quotation marks (""") containing the receiver encoded in the ANSI character set of the default code page for the current locale.

If the length of the receiver is zero (0), the string "<null>" is returned.

**firstCharToLower**

**Signature**  
`firstCharToLower() updating;`

The `firstCharToLower` method of the `StringUtf8` primitive type converts an uppercase first character in the receiving string to lowercase, according to the conventions of the current locale.

The following example shows the use of the `firstCharToLower` method.

```pascal
vars
    str8 : StringUtf8;
begin
    str8 := '@HELLO WORLD';
    str8.firstCharToLower;
    write str8;      // Outputs 'hELLO WORLD'
end;
```
**firstCharToUpper**

**Signature**

```plaintext
firstCharToUpper() updating;
```

The `firstCharToUpper` method of the `StringUtf8` primitive type converts a lowercase first character in the receiving string to uppercase, according to the conventions of the current locale. The following example shows the use of the `firstCharToUpper` method.

```plaintext
vars
  str8: StringUtf8;
begin
  str8 := @'hello world';
  str8.firstCharToUpper;
  write str8;       // Outputs 'Hello world'
end;
```

**isValid**

**Signature**

```plaintext
isValid(): Boolean;
```

The `isValid` method of the `StringUtf8` primitive type returns `true` if the receiver is a correctly formatted UTF8 string.

**length**

**Signature**

```plaintext
length(): Integer;
```

The `length` method of the `StringUtf8` primitive type returns the current number of characters of the receiver. The following example shows the use of the `length` method.

```plaintext
vars
  str8: StringUtf8;
begin
  str8 := @"hello world";
  write str8.length; // Outputs 11
end;
```

**maxLength**

**Signature**

```plaintext
maxLength(): Integer;
```

The `maxLength` method of the `StringUtf8` primitive type returns the declared maximum length of the receiver. If the maximum length of a `StringUtf8` variable has not been declared, -1 is returned.

The following example shows the use of the `maxLength` method.

```plaintext
vars
  str8: StringUtf8[100];
begin
  str8 := @"hello world";
  write str8.maxLength; // Outputs 100
end;
```
**padBlanks**

**Signature**  
`padBlanks(int: Integer): StringUtf8;`

The `padBlanks` method of the `StringUtf8` primitive type returns a string of the length specified in the `int` parameter, consisting of the receiving string padded with appended (trailing) spaces.

If the string is longer than the integer value, it is not truncated but the whole string is returned.

The following example shows the use of the `padBlanks` method.

```
vars
str8 : StringUtf8;
begin
  str8 := @'Alfonso: '; 
  write str8.padBlanks(10) & '123 Sesame St.';
  // Outputs 'Alfonso: 123 Sesame St.'
end;
```

**padLeadingZeros**

**Signature**  
`padLeadingZeros(int: Integer): StringUtf8;`

The `padLeadingZeros` method of the `StringUtf8` primitive type returns a string of the length specified in the `int` parameter, consisting of the receiving string padded with leading zeros.

The following example shows the use of the `padLeadingZeros` method.

```
vars
str8 : StringUtf8;
begin
  str8 := @'123.45';
  write str8.padLeadingZeros(10);  // Outputs '0000123.45'
end;
```

**pos**

**Signature**  
`pos(substr: StringUtf8;  
  start: Integer): Integer;`

The `pos` method of the `StringUtf8` primitive type returns an integer containing the character index of the start of a substring within a string. The substring is specified by the `substr` parameter. The search for the substring begins at the character index specified by the `start` parameter.

The `start` parameter must be greater than zero (0) and less than or equal to the length of the receiver. If the `substr` or the `start` parameter is greater than the length of the receiver, this method returns zero (0).

This method returns zero (0) if the specified substring is not found.

**Note**  
The character search is case-sensitive.

The following example shows the use of the `pos` method.

```
vars
str8 : StringUtf8;
begin
  str8 := @'position example';
```
posUsingByteOffset

Signature  posUsingByteOffset(substr: StringUtf8; start: Integer): Integer;

The posUsingByteOffset method of the StringUtf8 primitive type returns an integer containing the byte offset of the start of a UTF8 substring within the receiver. The substring is specified by the substr parameter.

The search for the substring begins at the byte offset specified by the start parameter.

The start parameter must be greater than zero (0) and less than or equal to the number of bytes in the receiver. If the substr or the start parameter is greater than the number of bytes in the receiver, this method returns zero (0).

This method returns zero (0) if the specified substring is not found.

Note  The character search is case-sensitive.

In the following code example, the two characters of the string str8 require three bytes and two bytes in the UTF8 encoding. The first character starts at offset one (1) and the second character at offset four (4).

vars  str8 : StringUtf8;
begin  str8 := "۩@©";
  write str8.posUsingByteOffset("©", 3);  // Outputs 4
end;

replaceChar

Signature  replaceChar(char: Character; withChar: Character): updating;

The replaceChar method of the StringUtf8 primitive type replaces all occurrences of the character specified in the char parameter with the character specified in the withChar parameter.

Note  The character replacement is case-sensitive.

The following example shows the use of the replaceChar method.

vars  str8 : StringUtf8;
begin  str8 := "zhis example shows character replacement";
  write str8;  // Outputs: zhis example shows character replacement
  str8.replaceChar("z", "T");
  write str8;  // Outputs: This example shows character replacement
end;
reverse

Signature reverse(): StringUtf8;

The reverse method of the StringUtf8 primitive type returns a string consisting of the receiving string with the position of all characters reversed. For example, a string that contains "abcde" is returned as "edcba".

The following example shows the use of the reverse method.

```pascal
vars
   str8 : StringUtf8;
begin
   str8 := 'abcde';
   write str8.reverse;   // Outputs 'edcba'
end;
```

reversePos

Signature reversePos(substr: StringUtf8): Integer;

The reversePos method of the StringUtf8 primitive type returns the position of the last occurrence of the substring specified in the substr parameter in the receiving string.

Note The character search is case-sensitive.

The following example shows the use of the reversePos method.

```pascal
vars
   str8 : StringUtf8;
begin
   str8 := "Reverse position example";
   write str8.reversePos('pos');   // Outputs 9
end;
```

reversePosIndex

Signature reversePosIndex(substr: StringUtf8; index: Integer): Integer;

The reversePosIndex method of the StringUtf8 primitive type returns the position of the last occurrence of the substring specified in the substr parameter, in a string formed from the first character of the receiving string up to (and including) the character position specified in the index parameter.

Notes The character search is case-sensitive.

The value of the index parameter cannot exceed the length of the receiving string.

The following example shows the use of the reversePosIndex method.

```pascal
vars
   str8 : StringUtf8;
   count : Integer;
begin
   str8 := "car->taxi->bus->train";
   count := str8.length;
   while count > 0 do
```
Encyclopedia of Primitive Types

StringUtf8 Type

count := str8.reversePosIndex('>', count);
write count;
count := count - 1;
endwhile;
// Outputs 15
// Outputs 10
// Outputs 4
// Outputs 0
end;

This method returns zero (0) if the specified substring is not found.

scanUntil

Signature  scanUntil(delimiters: StringUtf8;
       index:   Integer io): StringUtf8;

The scanUntil method of the StringUtf8 primitive type returns a UTF8 substring of the receiving string starting from the index specified in the index parameter up to (but not including) the first occurrence of any of the characters specified in the delimiters parameter.

The index of the delimiting character is returned in the second parameter. If a delimiting character is not found, the return value is the remainder of the receiving string (from the specified index) and an index value of zero (0) is returned in the second parameter.

Note  The character search is case-sensitive.

The following example shows the use of the scanUntil method.

vars
   str8 : StringUtf8;
   pos : Integer;
begin
   str8 := @"this:is/a;string";
pos := 1;
   write str8.scanUntil(@":/;", pos); // Outputs this
   pos := pos + 1;
   write str8.scanUntil(@":/;", pos); // Outputs is
end;

scanWhile

Signature  scanWhile(characters: StringUtf8;
       index:   Integer io): StringUtf8;

The scanWhile method of the StringUtf8 primitive type returns a UTF8 substring of the receiving string starting from the index specified in the index parameter up to (but not including) the first occurrence of any character other than the characters specified in the characters parameter.

The index of the delimiting character is specified in the second parameter. If a delimiting character is not found, the return value is the remainder of the string and an index value of zero (0) is returned in the second parameter, as shown in the following example.

vars
   index : Integer;
   str8 : StringUtf8;
StringUtf8 Type

begin
  index := 3;
  str8 := @'0246'.scanWhile(@'0123456789', index);
  write '<' & str8 & '>' & index.StringUtf8;
  // outputs <46> 0, not <> 0
end;

Notes   The character search is case-sensitive.
The delimiting character is any character that is not specified in the characters parameter.

The following example shows the use of the scanWhile method.

vars
  str8 : StringUtf8;
  index : Integer;
begin
  str8 := @"this:is/a:string";
  index := 1;
  write str8.scanWhile(@"abcdefghijklmnopqrstuvwxyz", index);
  // Outputs this
  index := index + 1;
  write str8.scanWhile(@"abcdefghijklmnopqrstuvwxyz", index);
  // Outputs is
end;

size

Signature  size(): Integer;

The size method of the StringUtf8 primitive type returns the number of bytes required to store the receiver. Note that this value does not include the null character that marks the end of the string.

The following example shows the use of the size method.

vars
  str8 : StringUtf8;
beg
  str8 := @"JADE";
  write str8.size;   // 4 bytes - one for each ASCII character
  str8 := @"&copy;";
  write str8.size;   // 2 bytes for the copyright symbol
  str8 := @"€";
  write str8.size;   // 3 bytes for the euro currency symbol
end;

substringAtByteOffset

Signature  substringAtByteOffset(offset: Integer;
                                      length: Integer): StringUtf8;

The substringAtByteOffset method of the StringUtf8 primitive type returns a UTF8 substring beginning with the character that starts at the byte offset specified by the value of the offset parameter within the receiving UTF8 string or after that offset; that is, the method scans from the offset position forwards to find the next character.
The value of the length parameter determines the maximum number of characters that can be returned in the UTF8 substring.

In the following code example, the first character of the string str8 requires three bytes for UTF8 encoding. The first character starts at byte offset one (1) and the second character at byte offset four (4).

```plaintext
vars
    str8: StringUtf8;
begin
    str8 := @"€xyz";
    write str8.substringAtByteOffset(3,2); // writes xy
end;
```

toLower

**Signature**  
toLower(): StringUtf8;

The **toLower** method of the **StringUtf8** primitive type returns a copy of the receiving string with all uppercase characters converted to lowercase, according to the conventions of the current locale.

The following example shows the use of the **toLower** method.

```plaintext
vars
    str8: StringUtf8;
begin
    str8 := "UPPERCASE TEXT CAN LOOK THREATENING";
    write str8.toLowerCase;
    // Outputs uppercase text can look threatening
end;
```

toUpper

**Signature**  
toUpper(): StringUtf8;

The **toUpper** method of the **StringUtf8** primitive type returns a copy of the receiving string with all lowercase characters converted to uppercase, according to the conventions of the current locale.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The following example shows the use of the **toUpper** method.

```plaintext
vars
    str8: StringUtf8;
begin
    str8 := "lowercase";
    write str8.toUpperCase;  // Outputs LOWERCASE
end;
```
trimBlanks

**Signature**  trimBlanks(): StringUtf8;

The **trimBlanks** method of the **StringUtf8** primitive type returns a copy of the receiving string with blanks (spaces) trimmed from both ends of the receiver.

The following example shows the use of the **trimBlanks** method.

```pascal
vars
  str8 : StringUtf8;
begin
  str8 := ' some text ';
  write str8.trimBlanks;   // Outputs 'some text'
end;
```

trimLeft

**Signature**  trimLeft(): StringUtf8;

The **trimLeft** method of the **StringUtf8** primitive type returns a copy of the receiving string with leading blanks (spaces) removed.

The following example shows the use of the **trimLeft** method.

```pascal
vars
  str8 : StringUtf8;
begin
  str8 := ' some text ';
  write str8.trimLeft;    // Outputs 'some text'
end;
```

trimRight

**Signature**  trimRight(): StringUtf8;

The **trimRight** method of the **StringUtf8** primitive type returns a copy of the receiving string with trailing blanks (spaces) trimmed from the end of the receiver.

The following example shows the use of the **trimRight** method.

```pascal
vars
  str8 : StringUtf8;
begin
  str8 := ' some text ';
  write str8.trimRight;  // Outputs ' some text'
end;
```
Time Type

Use the Time primitive type to declare a variable representing the time of day since midnight to the nearest millisecond.

If you declare a Time primitive type variable in your method, it is initialized with the current time each time the method that declares the variable is invoked. Object properties of Time primitive type are initialized to null.

In JADE thin client mode, local variables of type Time are always initialized to the time relative to the presentation client.

The following example shows the use of the Time primitive type.

```pascal
testTime();
vars
time : Time;
h,m,s,ms : Integer;
begin
  h := 15;
m := 39;
s := 06;
ms := 45;
time.setTime(h, m, s, ms);
write time; // Outputs 15:39:06
end;
```

The following table lists valid operations for the Time primitive type.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expression Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>time-expression + integer-expression</td>
<td>(time)</td>
</tr>
<tr>
<td>time-expression - integer-expression</td>
<td>(time)</td>
</tr>
<tr>
<td>time-expression + time-expression</td>
<td>(time)</td>
</tr>
<tr>
<td>time-expression - time-expression</td>
<td>(integer)</td>
</tr>
<tr>
<td>time-expression + timestamp-expression</td>
<td>(timestamp)</td>
</tr>
</tbody>
</table>

The following example, in which 10 minutes is added to a Time primitive type variable, uses a millisecond integer value.

```pascal
  time := time + 600000 // 10 * 60 * 1000
```

If you add 60000 milliseconds (one minute) to a time variable and the assigned time is later than 23:59:59:999, the resulting value is 00:00:59:999 or later.

For details about the methods defined in the Time primitive type, see "Time Methods", in the following subsection.

For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the JADE Developer's Reference.
### Time Methods

The methods defined in the **Time** primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>currentLocaleFormat</td>
<td>Returns a string containing the time in the format of the current locale</td>
</tr>
<tr>
<td>display</td>
<td>Returns a string representing the value of the receiver</td>
</tr>
<tr>
<td>format</td>
<td>Returns a string containing the time in the specified format</td>
</tr>
<tr>
<td>hour</td>
<td>Returns the hour part of the receiver in 24-hour clock form</td>
</tr>
<tr>
<td>isValid</td>
<td>Returns <strong>true</strong> if the receiver contains a valid time</td>
</tr>
<tr>
<td>milliSecond</td>
<td>Returns the millisecond part of the receiver</td>
</tr>
<tr>
<td>minute</td>
<td>Returns the minute part of the receiver</td>
</tr>
<tr>
<td>parseWithCurrentLocale</td>
<td>Sets the receiver to the result of parsing a string representing a time for the current locale</td>
</tr>
<tr>
<td>parseWithFmtAndLcid</td>
<td>Sets the receiver to the result of parsing a string representing a time for the specified format and the specified locale</td>
</tr>
<tr>
<td>parseWithPicAndLcid</td>
<td>Sets the receiver to the result of parsing a string representing a time for the specified time picture and the specified locale</td>
</tr>
<tr>
<td>second</td>
<td>Returns the second part of the receiver</td>
</tr>
<tr>
<td>setByteOrderLocal</td>
<td>Returns a time that has the bytes ordered as required by the local node</td>
</tr>
<tr>
<td>setByteOrderRemote</td>
<td>Returns a time that has the bytes ordered as required by the specified remote node</td>
</tr>
<tr>
<td>setTime</td>
<td>Sets the value of the receiver to a specified time in 24-hour clock form</td>
</tr>
<tr>
<td>setTimeStrict</td>
<td>Sets the value of the receiver to a specified time and checks that the individual time values are within range</td>
</tr>
<tr>
<td>subtract</td>
<td>Returns the interval between the receiver and the specified time</td>
</tr>
<tr>
<td>userFormat</td>
<td>Returns a string containing the receiver in the supplied time format</td>
</tr>
<tr>
<td>userFormatAndLcid</td>
<td>Returns a string containing the receiver in the specified time format for the specified locale</td>
</tr>
<tr>
<td>userFormatPicAndLcid</td>
<td>Returns a string containing the receiver in the specified time picture for the specified locale</td>
</tr>
</tbody>
</table>

#### currentLocaleFormat

**Signature**  
currentLocaleFormat(): String;

The **currentLocaleFormat** method of the **Time** primitive type returns a string containing the time in the format of the current locale.

If you do not define the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.
display

Signature  display(): String;

The display method of the Time primitive type returns a string representing the value of the receiver.

format

Signature  format(picture: String): String;

The format method of the Time primitive type returns a string containing the time in the format specified in the picture parameter and current locale settings for time markers (AM/PM). For example:

```javascript
testTimeFormat();
vars
time : Time;
begin
write "The time now is " & time.format("hh.m:ss tt");
end;
```

The example shown in this method writes The time now is 08.41:08 a.m. (if the Time regional setting for that user locale has the AM symbol specified as a.m.).

Use the string picture elements listed in the following table to construct time format picture strings. Separate each element with a space or a separator character; for example, a period (.) or a colon character (:

<table>
<thead>
<tr>
<th>Picture</th>
<th>Description</th>
<th>Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>Hours, with no leading zero (12-hour clock)</td>
<td>8</td>
</tr>
<tr>
<td>hh</td>
<td>Hours, with a leading zero (12-hour clock)</td>
<td>08</td>
</tr>
<tr>
<td>H</td>
<td>Hours, with no leading zero (24-hour clock)</td>
<td>13</td>
</tr>
<tr>
<td>HH</td>
<td>Hours, with a leading zero (24-hour clock)</td>
<td>08</td>
</tr>
<tr>
<td>m</td>
<td>Minutes, with no leading zero</td>
<td>6</td>
</tr>
<tr>
<td>mm</td>
<td>Minutes, with leading zero</td>
<td>06</td>
</tr>
<tr>
<td>s</td>
<td>Seconds, with no leading zero</td>
<td>47</td>
</tr>
<tr>
<td>ss</td>
<td>Seconds, with leading zero</td>
<td>07</td>
</tr>
<tr>
<td>t</td>
<td>One-character time marker string of the current locale</td>
<td>p</td>
</tr>
<tr>
<td>tt</td>
<td>Multiple-character time marker string of the current locale</td>
<td>PM</td>
</tr>
</tbody>
</table>

In this table, the t and tt picture elements are determined by the AM symbol or PM symbol for the current locale of the user (defined by using the AM symbol or PM symbol combo box in the Time sheet of the Regional Settings Properties dialog, accessed from the Regional Settings icon in the Control Panel).
Notes You can use the `defineTimeFormat` method of the `TimeFormat` class if you want to create your own transient format objects and define a time format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

If you do not define the `EnhancedLocaleSupport` parameter in the `[JadeEnvironment]` section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

## hour

**Signature**  
`hour(): Integer;`

The `hour` method of the `Time` primitive type returns the hour part of the receiver in 24-hour clock form.

The following example shows the use of the `hour` method.

```pascal
vars  
  h, m, s : Integer;
begin
  // Call this method to alter the time settings
  displayedTime.setTime(displayedTime.hour + h,  
       displayedTime.minute + m,  
       displayedTime.second + s, 00);
  clockFrame.caption := displayedTime.String;
end;
```

## isValid

**Signature**  
`isValid(): Boolean;`

The `isValid` method of the `Time` primitive type returns true if the receiver contains a valid time value.

The code fragment in the following example shows the use of the `isValid` method.

```pascal
if not any.Time.isValid() then
  app.msgBox("New value must contain a valid Time", "No date entered",  
       MsgBox_OK_Only);
  return false;
endif;
```

Use this method after a conversion instead of testing for a null value, as null indicates midnight, which is a valid time.

## milliSecond

**Signature**  
`milliSecond(): Integer;`

The `milliSecond` method of the `Time` primitive type returns the millisecond part of the receiver.

The code fragment in the following example shows the use of the `milliSecond` method.

```pascal
if eventTag = 3 then
  displayedTime.setTime(displayedTime.hour,  
                   displayedTime.minute,  
                   displayedTime.milliSecond + s, 00);
  clockFrame.caption := displayedTime.String;
```
minute

**Signature** minute(): Integer;

The `minute` method of the `Time` primitive type returns the minute part of the receiver.

For an example of the use of the `minute` method, see the `Time` primitive type `hour` method.

**parseWithCurrentLocale**

**Signature** parseWithCurrentLocale(source: String; errOffset: Integer output): Integer updating;

The `parseWithCurrentLocale` method of the `Time` primitive type parses the string specified in the `source` parameter to ensure that it matches the time format of the current locale in terms of element (milliseconds, seconds, minutes, hours) order and separators.

If the value of the `source` parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output `errOffset` parameter, and sets the receiver to the invalid time value (the `isValid` method of the `Time` primitive type will return `false`).

This is equivalent to calling the `parseWithFmtAndLcid` method, passing null in the `fmt` parameter and zero (0) in the `lcid` parameter.

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseWithFmtAndLcid**

**Signature** parseWithFmtAndLcid(source: String; fmt: TimeFormat; lcid: Integer; errOffset: Integer output): Integer updating;

The `parseWithFmtAndLcid` method of the `Time` primitive type parses the string specified in the `source` parameter to ensure that it matches the time format specified in the `fmt` parameter.

The time must match the appropriate values for the locale specified in the `lcid` parameter. If the source string contains a valid time, it is assigned to the receiver; otherwise the invalid time value is assigned to the receiver (the `isValid` method of the `Time` primitive type will return `false`).
If the value of the fmt parameter is null, the time format of the locale specified in the lcid parameter is used. If the value of the lcid parameter is zero (0), the time format of the current locale is used. If the value of the fmt parameter is not null, the AM/PM indicators, if specified, are used rather than the locale indicators.

If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output errOffset parameter, and sets the receiver to the invalid time value.

Leading zeros in the hour, minute, and second elements are optional.

This method is the same as the parseWithPicAndLcid method except that the picture string is taken from the TimeFormat class format property. For more details and examples of valid date matches, see the parseWithPicAndLcid method.

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**parseWithPicAndLcid**

**Signature**

```java
parseWithPicAndLcid(source: String;
pic: String;
lcid: Integer;
errOffset: Integer output): Integer updating;
```

The parseWithPicAndLcid method of the Time primitive type parses the string specified in the source parameter using the specified time picture and locale, validating that the source matches the time format picture to ensure that it matches the time picture string specified in the pic parameter.

If the source string contains a valid time, it is assigned to the receiver; otherwise the invalid time value is assigned to the receiver (the isValid method of the Time primitive type will return false).

If the value of the pic parameter is null, the time format picture of the locale specified in the lcid parameter is used. If the value of the lcid parameter is zero (0), the time format picture of the current locale is used.

If the value of the source parameter matches the format rules, the method returns zero (0) and sets the receiver to the parsed value. If it does not match the format rules, it returns a JADE error code (parse errors are in the range 1800 through 1869), indicates the first offending character returning its zero-based offset using the output errOffset parameter, and sets the receiver to the invalid time value.

Leading zeros in the hour, minute, and second elements are optional when separators are specified.

If the marker picture is "t", the source marker text must be a single character matching the first character of one of the AM/PM indicators for the locale. If the marker picture is "tt" (or longer), the source marker text must match exactly one of the AM/PM indicators for the locale. A locale-based case-insensitive comparison is used. If the AM indicator for the locale is "a.m." (for example, New Zealand), the indicator "AM" (for example, United States) is also accepted.

If the hour picture is "h" or "hh", the hour value must be in the range 0 through 12. If the hour picture is "H" or "HH", the hour value must be in the range 0 through 23. The minute and second values must be in the range 0 through 59.

Source text "12:00a.m." and "00:00a.m." with picture "h:mmmtt" or "hh:mmmtt" converts to time **00:00** (midnight). Source text "12:00p.m." with picture "h:mmmtt" or "hh:mmmtt" converts to time **12:00** (midday).
The format can include ".fff" after ".s", to recognize a millisecond value. The decimal separator for the locale is expected between the second and millisecond values. The text can include zero (0) through three digits in the millisecond value.

The "H:mm:ss.fff" picture allows the following.

- "1:23:45.678" 1:23:45.678
- "1:23:45" 1:23:45.000
- "1:23:45.6" 1:23:45.600
- "1:23:45.600" 1:23:45.600
- "1:23:45.006" 1:23:45.006
- "1:23:45.070" 1:23:45.070

If you do not define the EnhancedLocaleSupport parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**second**

**Signature**

```java
second(): Integer;
```

The second method of the Time primitive type returns the second part of the receiver.

For an example of the use of the second method, see the Time primitive type hour method.

**setByteOrderLocal**

**Signature**

```java
setByteOrderLocal(architecture: Integer): Time;
```

The setByteOrderLocal method of the Time primitive type returns a time that has the bytes ordered as required by the local node.

The bytes of the receiver are assumed to be ordered as indicated by the architecture parameter.

The architecture parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the getOSPlatform method of the Node class.

The architecture can be one of the Node class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>
setByteOrderRemote

**Signature**

`setByteOrderRemote(architecture: Integer): Time;`

The `setByteOrderRemote` method of the `Time` primitive type returns a time that has the bytes ordered as required by the remote node indicated by the `architecture` parameter.

The bytes of the receiver are assumed to be ordered as required by the local node.

The `architecture` parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the `getOSPlatform` method of the `Node` class.

The architecture can be one of the `Node` class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>

**Note** This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

setTime

**Signature**

`setTime(hours: Integer; minutes: Integer; seconds: Integer; milliseconds: Integer): Boolean updating;`

The `setTime` method of the `Time` primitive type sets the value of the receiver to a specified time in 24-hour clock form using any valid combination of parameters. This method returns `true` if the specified time is valid or it returns `false` if it is invalid (for example, 24:00).

The parameters are integer values for hours, minutes, seconds, and milliseconds.

The code fragments in the following examples show the use of the `setTime` method.

```plaintext
startTime.setTime(startH.Integer, startM.Integer, 0, 0);
endTime.setTime(endH.Integer, endM.Integer, 0, 0);

if stopWatchButton.caption = "Stop Watch" then
  endTimer(1);
  stopWatchButton.caption := "Start";
  app.doWindowEvents (1);
  clockFrame.caption := "00:00:00";
```
You can use this method to set a valid time less than 24 hours in milliseconds. For example, the following code fragment sets the time to twelve hours in milliseconds.

```plaintext
displayedTime.setTime(0, 0, 0, 0);
elseif ... then
  ...
endif;
```

You can use the `displayedTime` method to set a valid time less than 24 hours in milliseconds. For example, the following code fragment sets the time to twelve hours in milliseconds.

```plaintext
startTime.setTime(0, 720, 0, 0);
```

### set_time_strict

**Signature**

```plaintext
setTimeStrict(hours: Integer;
minutes: Integer;
seconds: Integer;
milliseconds: Integer): Boolean updating;
```

The `setTimeStrict` method of the `Time` primitive type checks that the hours, minutes, seconds, and milliseconds specified in the method parameters are within the range of time in the `HH:MM:SS:sss` (24-hour clock form) format.

If any of the specified parameters is outside the range of the `HH:MM:SS:sss` time format (that is, a value that is greater than 23 hours, 59 minutes, 59 seconds, or 999 milliseconds), this method returns `false` and sets the receiver to the specified "invalid" time.

**Tip** Use the `Time` primitive type `setTime` method to set the value of the receiver in 24-hour clock form to a specified time using any valid combination of parameters.

### subtract

**Signature**

```plaintext
subtract(time: Time): TimeStampInterval;
```

The `subtract` method of the `Time` primitive type returns the interval between the receiver and the value of the `time` parameter as a `TimeStampInterval` value.

The following example shows the use of the `subtract` method.

```plaintext
vars
  now, hourFromNow : Time;
begin
  hourFromNow:= now + 60 * 60 * 1000;
  write hourFromNow.subtract(now); // Outputs "0:01:00:00.000"
end;
```

### user_format

**Signature**

```plaintext
userFormat(fmt: TimeFormat): String;
```

The `userFormat` method of the `Time` primitive type returns a string containing the receiver in the specified time format.

To define your time formats, use the Schema menu `Format` command from the Schema Browser.
Notes  When you use a format in a JADE method, prefix your user time format name with a dollar sign ($); for example, `userFormat($MyTime)`.

You can use the `defineTimeFormat` method of the `TimeFormat` class if you want to create your own transient format objects and define a time format that dynamically overrides the format for the locale at run time. (For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.)

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The code fragments in the following examples show the use of the `userFormat` method.

```java
tblTime.text := p.name & " " &
    p.startTime.userFormat($HourMin) & "-" &
    p.endTime.userFormat($HourMin) & " ");
if counter > 0 then
    igfFrame.myOutline.IGOutline.addXLabel(time.userFormat
        ($PlainTime));
endif;
```

**userFormatAndLcid**

**Signature**  
`userFormatAndLcid(fmt: TimeFormat; lcid: Integer): String;`

This `userFormatAndLcid` method of the `Time` primitive type returns a string containing the receiver in the time format specified in the `fmt` parameter of the locale specified in the `lcid` parameter.

If the value of the `fmt` parameter is null, the time format of the locale specified in the `lcid` parameter is returned. If the value of the `lcid` parameter is zero (0), the time format of the current locale is returned, same as the `userFormatPicAndLcid` method except that the picture string is taken from the `TimeFormat` class format property. For more details and examples of valid date matches, see the `userFormatPicAndLcid` method.

If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to false, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

**userFormatPicAndLcid**

**Signature**  
`userFormatPicAndLcid(pic: String; lcid: Integer): String;`

This `userFormatPicAndLcid` method of the `Time` primitive type returns a string containing the receiver in the time format picture specified in the `pic` parameter of the locale specified in the `lcid` parameter.

If the value of the `pic` parameter is null, the time format picture of the locale specified in the `lcid` parameter is returned. If the value of the `lcid` parameter is zero (0), the time format picture of the current locale is returned.

The picture string can include ".fff" following "s", to output the millisecond part of the time; for example, "H:mm:ss.fff" can generate the strings "13:07:23.543" and "9:53:11.000".
**Note** If you do not define the `EnhancedLocaleSupport` parameter in the [JadeEnvironment] section of the JADE initialization file on the database node or you set it to `false`, inconsistent results could be returned to the application server when running in JADE thin client mode and there are locale overrides, as all overrides on the application server are suppressed if enhanced locale support is not enabled.

The following example of the `userFormatPicAndLcid` method outputs **15:07:23.123**.

```plaintext
vars
t : Time;
s : String;
begin
t.setTime(15,7,23,123);
s := t.userFormatPicAndLcid("HH:mm:ss.fff", 0);
write s;
end;
```
TimeStamp Type

A TimeStamp primitive type is used to store the variable as type timestamp; that is, the date and time. If you declare a variable of type TimeStamp in your method, it is initialized with the current date and time each time the method is invoked.

In JADE thin client mode, local variables of type TimeStamp are always initialized to the date and time relative to the presentation client.

The following example shows the use of the TimeStamp primitive type.

```pascal
vars
    timeStamp : TimeStamp;
    time : Time;
    h,m,s,ms : Integer;
begin
    h := 15;
    m := 39;
    s := 06;
    time.setTime(h, m, s, ms);
    timeStamp.setTime(time); // Assigns format to string
    write timeStamp; // Outputs 11 August 2000 3:39pm
end;
```

The following table lists valid operations for the TimeStamp primitive type.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expression Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp-expression + time-expression</td>
<td>(timestamp)</td>
</tr>
<tr>
<td>timestamp-expression - time-expression</td>
<td>(timestamp)</td>
</tr>
<tr>
<td>timestamp-expression + timestampinterval-expression</td>
<td>(timestamp)</td>
</tr>
<tr>
<td>timestamp-expression - timestampinterval-expression</td>
<td>(timestamp)</td>
</tr>
</tbody>
</table>

**Caution** The TimeStamp value that results from adding to a timestamp value or subtracting from a timestamp value does not take daylight saving into account.

For details about the methods defined in the TimeStamp primitive type, see "TimeStamp Methods", in the following subsection. For details about converting primitive types, see "Converting Primitive Types", in Chapter 1 of the JADE Developer’s Reference.

**TimeStamp Methods**

The methods defined in the TimeStamp primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Returns the date part of the receiver</td>
</tr>
<tr>
<td>display</td>
<td>Returns a string representing the value of the receiver</td>
</tr>
<tr>
<td>isValid</td>
<td>Returns <code>true</code> if the receiver contains a valid timestamp value</td>
</tr>
<tr>
<td>literalFormat</td>
<td>Returns a string representing the receiver in literal format</td>
</tr>
</tbody>
</table>
TimeStamp Type

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>localToUTCTime</td>
<td>Converts a timestamp in local time to UTC using the time zone in which the method executes</td>
</tr>
<tr>
<td>localToUTCTimeUsingBias</td>
<td>Converts a timestamp in local time to UTC using the specified bias</td>
</tr>
<tr>
<td>setByteOrderLocal</td>
<td>Returns a timestamp that has the bytes ordered as required by the local node</td>
</tr>
<tr>
<td>setByteOrderRemote</td>
<td>Returns a timestamp that has the bytes ordered as required by the specified remote node</td>
</tr>
<tr>
<td>setDate</td>
<td>Sets the date part of the receiver to a specified date</td>
</tr>
<tr>
<td>setTime</td>
<td>Sets the time part of the receiver to a specified time</td>
</tr>
<tr>
<td>time</td>
<td>Returns the time part of the receiver</td>
</tr>
<tr>
<td>utcToLocalTime</td>
<td>Converts a timestamp in UTC time to local time for the time zone in which the method executes</td>
</tr>
<tr>
<td>utcToLocalTimeUsingBias</td>
<td>Converts a timestamp in UTC time to local time using the specified bias</td>
</tr>
</tbody>
</table>

**date**

**Signature**

date(): Date;

The `date` method of the `TimeStamp` primitive type returns a date that is the same as the date part of the receiver.

**Note** If you change the date returned by the `date` method using the `setDate` method of the `Date` primitive type, you are not actually changing the date part of the timestamp.

```javascript
vars
ts : TimeStamp;
begin
    ts.date.setDate(1,2,2007); // does not change ts
```

To change the date part of a timestamp, use the `setDate` method of the `TimeStamp` primitive type:

```javascript
vars
ts : TimeStamp;
d : Date;
begin
d.setDate(1,2,2007);
ts.setDate(d); // does change ts
```

**display**

**Signature**
display(): String;

The `display` method of the `TimeStamp` primitive type returns a string representing the value of the receiver.

**isValid**

**Signature**
isValid(): Boolean;

The `isValid` method of the `TimeStamp` primitive type returns `true` if the receiver contains a valid timestamp value. This method returns `false` if the date or time is not valid.
**literalFormat**

**Signature**  
literalFormat(): String;

The **literalFormat** method of the **TimeStamp** primitive type returns a string representing the receiver in literal format.

The following example shows the use of the **literalFormat** method.

```plaintext
vars
    timeStamp : TimeStamp;
    date     : Date;
    time     : Time;
begin
    date.setDate(11, 5, 2013);
    time.setTime(16, 12, 23, 0);
    timeStamp.setDate(date);
    timeStamp.setTime(time);
    write timeStamp.literalFormat; // Outputs 2013:05:11:16:12:23

    date.setDate(11, 5, 2013);
    time.setTime(16, 12, 23, 5);
    timeStamp.setDate(date);
    timeStamp.setTime(time);
    write timeStamp.literalFormat; // Outputs 2013:05:11:16:12:23.005
end;
```

**localToUTCTime**

**Signature**  
localToUTCTime(): TimeStamp;

The **localToUTCTime** method of the **TimeStamp** primitive type converts a timestamp in local time to Coordinated Universal Time (UTC) using the time zone of the machine in which the method executes; for example, if the method is executing in an application server for a presentation client running on another machine, the bias is taken from the time zone of the machine running the application server.

To convert between local and UTC time in a thin client application in which you want to be sensitive to the bias of the presentation client machine, use the **localToUTCTimeUsingBias** method, as shown in the following code fragment.

```plaintext
    self.localToUTCTimeUsingBias(app.currentUTCBias(PresentationClient));
```

**Notes**  
Translations between UTC and local time are based on the formula UTC = local time + bias.

Greenwich Mean Time (GMT) has been replaced as the world standard time by Coordinated Universal Time (UTC), which is based on atomic measurements rather than the rotation of the earth. (GMT remains the standard time zone for the Prime Meridian, or zero longitude.)

See also the **TimeStamp** primitive type **localToUTCTimeUsingBias**, **utcToLocalTime**, and **utcToLocalTimeUsingBias** methods and the **Application** class **currentUTCBias** and **getUTCTime** methods.

**localToUTCTimeUsingBias**

**Signature**  
localToUTCTimeUsingBias(bias: Integer): TimeStamp;

The **localToUTCTimeUsingBias** method of the **TimeStamp** primitive type converts a timestamp in local time to UTC time using the number of minutes specified in the **bias** parameter.
**Note** Translations between UTC and local time are based on the formula $UTC = local\ time + bias$.

See also the `TimeStamp` primitive type `localToUTCTime`, `utcToLocalTimeUsingBias`, and `utcToLocalTime` methods and the `Application` class `currentUTCBias` and `getUTCTime` methods.

### setByteOrderLocal

**Signature**  
`setByteOrderLocal(architecture: Integer): TimeStamp;`

The `setByteOrderLocal` method of the `TimeStamp` primitive type returns a timestamp that has the bytes ordered as required by the local node.

The bytes of the receiver are assumed to be ordered as indicated by the `architecture` parameter.

The `architecture` parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the `getOSPlatform` method of the `Node` class.

The architecture can be one of the `Node` class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>

**Note** This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

### setByteOrderRemote

**Signature**  
`setByteOrderRemote(architecture: Integer): TimeStamp;`

The `setByteOrderRemote` method of the `TimeStamp` primitive type returns a timestamp that has the bytes ordered as required by the remote node indicated by the `architecture` parameter.

The bytes of the receiver are assumed to be ordered as required by the local node.

The `architecture` parameter is a unique number that indicates internal byte ordering and alignment information relevant to the hardware platform of this release of JADE and is returned by the `getOSPlatform` method of the `Node` class.

The architecture can be one of the `Node` class constant values listed in the following table.

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_32Big_Endian</td>
<td>32-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_32Little_Endian</td>
<td>32-bit little-endian internal byte ordering and alignment</td>
</tr>
</tbody>
</table>
**TimeStamp Type**

<table>
<thead>
<tr>
<th>Node Class Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture_64Big_Endian</td>
<td>64-bit big-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_64Little_Endian</td>
<td>64-bit little-endian internal byte ordering and alignment</td>
</tr>
<tr>
<td>Architecture_Gui</td>
<td>Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)</td>
</tr>
</tbody>
</table>

**Note** This method is not available on a Compact JADE node where it would result in a 1068 - Feature not available exception.

### setDate

**Signature**  
`setDate(date: Date) updating;`

The `setDate` method of the `TimeStamp` primitive type sets the date part of the receiver to a specified date.

### setTime

**Signature**  
`setTime(time: Time) updating;`

The `setTime` method of the `TimeStamp` primitive type sets the time part of the receiver to a specified time.

### time

**Signature**  
`time(): Time;`

The `time` method of the `TimeStamp` primitive type returns a time that is the same as the time part of the receiver.

**Note** If you change the time returned by the `time` method using the `setTime` method of the `Time` primitive type, you are not actually changing the time part of the timestamp.

```plaintext
vars
  ts : TimeStamp;
begin
  ts.time.setTime(09,10,11,999);  // does not change ts
```

To change the time part of a timestamp, use the `setTime` method of the `TimeStamp` primitive type:

```plaintext
vars
  ts : TimeStamp;
  t  : Time;
begin
  t.setTime(09,10,11,999);
  ts.setTime(t);  // does change ts
```

### utcToLocalTime

**Signature**  
`utcToLocalTime(): TimeStamp;`

The `utcToLocalTime` method of the `TimeStamp` primitive type converts a timestamp in UTC time to local time using the time zone of the machine in which the method executes; for example, if the method is executing in an application server for a presentation client running on another machine, the bias is taken from the time zone of the machine running the application server.
To convert between local and UTC time in a thin client application in which you want to be sensitive to the bias of the presentation client machine, use the `utcToLocalTimeUsingBias` method, as shown in the following code fragment.

```javascript
self.utcToLocalTimeUsingBias(app.currentUTCBias(PresentationClient));
```

**Note** Translations between UTC and local time are based on the formula \( UTC = local \text{ time} + bias \).

See also the `TimeStamp` primitive type `utcToLocalTimeUsingBias`, `localToUTCTimeUsingBias`, and `localToUTCTime` methods and the `Application` class `currentUTCBias` and `getUTCTime` methods.

**utcToLocalTimeUsingBias**

**Signature**

```javascript
utcToLocalTimeUsingBias(bias: Integer): TimeStamp;
```

The `utcToLocalTimeUsingBias` method of the `TimeStamp` primitive type converts a timestamp in UTC time to local time using the number of minutes specified in the `bias` parameter.

**Note** Translations between UTC and local time are based on the formula \( UTC = local \text{ time} + bias \).

See also the `TimeStamp` primitive type `utcToLocalTime`, `localToUTCTimeUsingBias`, and `localToUTCTime` methods and the `Application` class `currentUTCBias` and `getUTCTime` methods.
TimeStampInterval Type

The **TimeStampInterval** primitive type is used to represent the difference between two **TimeStamp** values. The **null** value for timestamp interval is equivalent to a timestamp interval of zero duration.

The following example shows a timestamp interval established by subtracting two timestamps. This example also shows the use of methods to display the number of whole days in an interval and the remaining time in milliseconds.

```plaintext
vars
time : Time;
date : Date;
ts1, ts2 : TimeStamp;
interval : TimeStampInterval;
begin
date.setDate(31,12,2007);       // New Year's Eve
time.setTime(23,59,0,0);        // Minute before midnight
ts1.setDate(date);
ts1.setTime(time);
date.setDate(1,1,2008);         // New Year's Day
time.setTime(12,0,0,0);         // Noon
ts2.setDate(date);
ts2.setTime(time);
interval := ts2 - ts1;
write interval;                  // 0:12:01:00.000 (days:hours:mins:secs)
write interval.getMilliseconds;  // 43260000 milliseconds
end;
```

The following table lists valid operations for the **TimeStampInterval** primitive type.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expression Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp-expression - timestamp-expression</td>
<td>(timestampinterval)</td>
</tr>
<tr>
<td>timestamp-expression + or - timestampinterval-expression</td>
<td>(timestamp)</td>
</tr>
<tr>
<td>timestampinterval-expression * or / integer-expression</td>
<td>(timestampinterval)</td>
</tr>
<tr>
<td>timestampinterval-expression + or - timestampinterval-expression</td>
<td>(timestampinterval)</td>
</tr>
<tr>
<td>timestampinterval-expression &lt; or &lt;= or = or &gt;= or &gt; or &lt;= timestampinterval-expression</td>
<td>(boolean)</td>
</tr>
</tbody>
</table>

**Caution** The **TimeStampInterval** value that results from subtracting two timestamp values does not take daylight saving into account.

For details about the methods defined in the **TimeStampInterval** primitive type, see "**TimeStampInterval Methods**", in the following subsection. For details about converting primitive types, see "**Converting Primitive Types**", in Chapter 1 of the *JADE Developer's Reference*. 
TimeStampInterval Methods

The methods defined in the **TimeStampInterval** primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display</td>
<td>Returns a string representing the value of the receiver</td>
</tr>
<tr>
<td>getMilliseconds</td>
<td>Returns the duration of the timestamp interval in milliseconds</td>
</tr>
<tr>
<td>isValid</td>
<td>Returns true if the receiver represents a valid timestamp interval</td>
</tr>
<tr>
<td>set</td>
<td>Sets the value of the receiver from a number of whole days and a number of milliseconds</td>
</tr>
</tbody>
</table>

**display**

**Signature**  
display(): String;

The **display** method of the **TimeStampInterval** primitive type returns a string representing the value of the receiver, in the format *days:hours:minutes:seconds.milliseconds*.

**getMilliseconds**

**Signature**  
getMilliseconds(): Integer64;

The **getMilliseconds** method of the **TimeStampInterval** primitive type returns the duration of the timestamp interval in milliseconds.

**isValid**

**Signature**  
isValid(): Boolean;

The **isValid** method of the **TimeStampInterval** primitive type returns true if the receiver contains a valid timestamp interval value.

**set**

**Signature**  
s set(days: Integer; milliseconds: Integer) updating;

The **set** method of the **TimeStampInterval** primitive type sets the value of a timestamp interval from the **days** and **milliseconds** parameters.

The value of the days and milliseconds parameters should both have the same sign. If one is positive and one is negative, an exception is raised.

If the value of the **milliseconds** parameter is greater than one day in milliseconds (that is, 86400000), the **TimeStampInterval** value that is set is incremented by the number of whole days that the **milliseconds** parameter value represents.
**TimeStampOffset Type**

The **TimeStampOffset** primitive type is used to represent a Coordinated Universal Time (UTC) date and time value, together with an offset that indicates how much that value differs from the local time when the value was set. The value of the offset component is precise to the minute.

A **TimeStampOffset** value unambiguously identifies a single point in time.

If you declare a **TimeStampOffset** primitive type local variable in your method, it is initialized with the current date, time, and offset from UTC each time the method that declares the variable is invoked. The offset is that of the presentation client if running in thin client mode; otherwise, it is the offset of the node where the code is executing.

The following example shows the initialization of a local **TimeStampOffset** variable.

```object
vars
tso : TimeStampOffset;
ts : TimeStamp;
begin
  // Executed in New Zealand, which is 13 hours 'ahead of Greenwich' in summer
  write ts; // outputs 20 January 2009, 09:15:20
  // (current date and time in New Zealand)
  write tso; // outputs 19 January 2009, 20:15:20 +1300
  // (current date and time in Greenwich)
end;
```

Object attributes of type **TimeStampOffset** are initialized with a null date, null time, and null offset; that is, **00:00:00 +0000**.

The following table lists valid operations for the **TimeStampOffset** primitive type.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expression Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestampoffset-expression + time-expression</td>
<td>(timestampoffset)</td>
</tr>
<tr>
<td>timestampoffset-expression - time-expression</td>
<td>(timestampoffset)</td>
</tr>
<tr>
<td>timestampoffset-expression + timestampinterval-expression</td>
<td>(timestampoffset)</td>
</tr>
<tr>
<td>timestampoffset-expression - timestampinterval-expression</td>
<td>(timestampoffset)</td>
</tr>
<tr>
<td>timestampoffset-expression - timestampoffset-expression</td>
<td>(timestampinterval)</td>
</tr>
<tr>
<td>timestampoffset-expression &lt; or &lt;= or = or &gt;= or &gt; or &lt;&gt; timestampoffset-expression</td>
<td>(boolean)</td>
</tr>
</tbody>
</table>

For details about the methods defined in the **TimeStampInterval** primitive type, see "**TimeStampOffset Methods**", in the following subsection. For details about converting primitive types, see "**Converting Primitive Types**", in Chapter 1 of the **JADE Developer’s Reference**.

**TimeStampOffset Methods**

The methods defined in the **TimeStampOffset** primitive type are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asLocalTimeStamp</td>
<td>Returns a timestamp representing the local time of the receiver</td>
</tr>
<tr>
<td>asUTCTimeStamp</td>
<td>Returns a timestamp representing the UTC time of the receiver</td>
</tr>
</tbody>
</table>
# Encyclopaedia of Primitive Types

## TimeStampOffset Type

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display</td>
<td>Returns a string representing the value of the receiver</td>
</tr>
<tr>
<td>getUTCBias</td>
<td>Returns the current bias in minutes</td>
</tr>
<tr>
<td>isValid</td>
<td>Returns true if the receiver represents a valid timestamp offset</td>
</tr>
<tr>
<td>setFromLocalTimeStamp</td>
<td>Sets the value of the receiver from the specified timestamp</td>
</tr>
</tbody>
</table>

### asLocalTimeStamp

**Signature**

```
asLocalTimeStamp(): TimeStamp;
```

The `asLocalTimeStamp` method of the `TimeStampOffset` primitive type returns a timestamp representing the local time of the receiver.

### asUTCTimeStamp

**Signature**

```
asUTCTimeStamp(): TimeStamp;
```

The `asUTCTimeStamp` method of the `TimeStampOffset` primitive type returns a timestamp representing the UTC time of the receiver.

### display

**Signature**

```
display(): String;
```

The `display` method of the `TimeStampOffset` primitive type returns a string representing the value of the receiver, in the format `date, time + offset`, where the offset is in the twenty-four hour clock format; for example, **19 January 2009, 02:56:10 +1300**.

### getUTCBias

**Signature**

```
getUTCBias(): Integer;
```

The `getUTCBias` method of the `TimeStampOffset` primitive type returns the current bias, which is the difference in minutes between Coordinated Universal Time (UTC) and local time (that is, `bias = UTC - 'local time'`).

As the bias is current, it includes any daylight saving adjustment in effect at the time the value is obtained.

### isValid

**Signature**

```
isValid(): Boolean;
```

The `isValid` method of the `TimeStampOffset` primitive type returns `true` if the date, time, and offset components of the receiver contain valid values.

### setFromLocalTimeStaMp

**Signature**

```
setFromLocalTimeStaMp(timeStamp: TimeStamp;
                      utcBias: Integer) updating;
```

The `setFromLocalTimeStaMp` method of the `TimeStampOffset` primitive type sets the value of the receiver from the `timeStamp` parameter and the offset specified in the `utcBias` parameter.
In the following example, a TimeStampOffset value is constructed from a local date and time together with an offset value obtained from the getUTCBias method.

```pascal
vars
tso_direct : TimeStampOffset;
bias : Integer;
tso_calculated : TimeStampOffset;
ts : TimeStamp;
begin
  write tso_direct;
bias := tso_direct.getUTCBias;
tso_calculated.setFromLocalTimeStamp(ts, bias);
  write tso_calculated;
end;
```
Global constants provide a more-meaningful representation than simply using literal values.

JADE provides system global constants at the Object class level, which are grouped by the following categories:

- ApplicationStatus
- CharacterConstants
- ColorConstants
- Environment
- Exceptions
- ExecutionLocation
- JadeDbFileVolatility
- JadeDynamicObjectName
- JadeDynamicObjectTypes
- JadeErrorCodeDatabase
- JadeErrorCodeIDE
- JadeErrorCodeRPS
- JadeErrorCodeSDS
- JadeErrorCodeWebService
- JadeLocaleIdNumbers
- JadeOdbc
- JadeProcessEvent
- JadeProfileString
- KeyCharacterCodes
- LockDurations
- LockTimeouts
- Locks
- MessageBox
- NotificationResponses
- ObjectVolatility
- PossibleTransientLeak
- Printer
- RPSTransitionHaltCode
For details about promoting class constants to global constants, see "Promoting Class Constants to Global Constants", in Chapter 4 of the JADE Development Environment User’s Guide.

**ApplicationStatus Category**

The global constants for the application mouse pointer (`app.mousePointer`) status are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busy</td>
<td>11</td>
<td>Hourglass mouse pointer indicates that the application is currently busy</td>
</tr>
<tr>
<td>Idle</td>
<td>0</td>
<td>Mouse pointer is in the idle state</td>
</tr>
</tbody>
</table>

For details about the mouse pointer (which controls the shape of the mouse pointer for all windows of the application), see the `Application` class `mousePointer` property in Chapter 1 of the JADE Encyclopaedia of Classes.

**CharacterConstants Category**

The global constants for the carriage return and tab characters are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>String or Character Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cr</td>
<td>&quot;0D&quot; character</td>
<td>Carriage return character</td>
</tr>
<tr>
<td>Crlf</td>
<td>&quot;0D 0A&quot; string</td>
<td>Carriage return / line feed characters</td>
</tr>
<tr>
<td>Lf</td>
<td>&quot;0A&quot; character</td>
<td>Line feed character</td>
</tr>
<tr>
<td>Tab</td>
<td>&quot;09&quot; character</td>
<td>Tab character</td>
</tr>
</tbody>
</table>
**ColorConstants Category**

The color global constants are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>0</td>
</tr>
<tr>
<td>Blue</td>
<td>16711680</td>
</tr>
<tr>
<td>DarkBlue</td>
<td>12582912</td>
</tr>
<tr>
<td>DarkGray</td>
<td>4210752</td>
</tr>
<tr>
<td>Gray</td>
<td>12632256</td>
</tr>
<tr>
<td>Green</td>
<td>32768</td>
</tr>
<tr>
<td>LightGreen</td>
<td>65280</td>
</tr>
<tr>
<td>LightYellow</td>
<td>8454143</td>
</tr>
<tr>
<td>Mauve</td>
<td>16711935</td>
</tr>
<tr>
<td>Purple</td>
<td>12583104</td>
</tr>
<tr>
<td>Red</td>
<td>255</td>
</tr>
<tr>
<td>White</td>
<td>16777215</td>
</tr>
<tr>
<td>Yellow</td>
<td>65535</td>
</tr>
</tbody>
</table>

**Environment Category**

The global constant for JADE environments that you can use in your applications, if required, is listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Primitive Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsUnicodeSystem</td>
<td>Boolean</td>
<td>(-1).Character.Integer &lt;&gt; 255</td>
</tr>
</tbody>
</table>

**Exceptions Category**

The global constants for exceptions are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex_Abort_Action</td>
<td>1</td>
<td>Causes the currently executing methods to be aborted</td>
</tr>
<tr>
<td>Ex_Continue</td>
<td>0</td>
<td>Resumes execution from the next expression after the expression that caused the exception</td>
</tr>
<tr>
<td>Ex_Pass_Back</td>
<td>-1</td>
<td>Passes control back to the prior local exception handler for this type of exception, or if a local handler is not found, a global exception handler for this type of exception</td>
</tr>
<tr>
<td>Ex_Resume_Next</td>
<td>2</td>
<td>Passes control back to the method that armed the exception handler</td>
</tr>
</tbody>
</table>

For more details, see "Exception Class Return Values", in Chapter 1 of the JADE Encyclopaedia of Classes.
**ExecutionLocation Category**

The global constants for the location of executed methods are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Method is executed…</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrentLocation</td>
<td>0</td>
<td>In the current location</td>
</tr>
<tr>
<td>DatabaseServer</td>
<td>1</td>
<td>On the database server node</td>
</tr>
<tr>
<td>PresentationClient</td>
<td>2</td>
<td>On the presentation client (applicable to applications running in thin client mode)</td>
</tr>
</tbody>
</table>

**JadeDbFileVolatility Category**

The global constants for the volatility of database files and partitions are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileVolatility_Frozen</td>
<td>Volatility_Frozen + 1</td>
</tr>
<tr>
<td>FileVolatility_Stable</td>
<td>Volatility_Stable + 1</td>
</tr>
<tr>
<td>FileVolatility_Transparent</td>
<td>0</td>
</tr>
<tr>
<td>FileVolatility_Volatile</td>
<td>Volatility_Volatile + 1</td>
</tr>
</tbody>
</table>

**JadeDynamicObjectNames Category**

The global constants for the name of dynamic objects used by JADE RootSchema classes are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>String Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JStats_ArrayName</td>
<td>&quot;JStatsArray&quot;</td>
</tr>
<tr>
<td>JStats_DictionaryName</td>
<td>&quot;JStatsDictionary&quot;</td>
</tr>
<tr>
<td>JStats_JadeBytesName</td>
<td>&quot;JStatsJadeBytes&quot;</td>
</tr>
<tr>
<td>JStats_setName</td>
<td>&quot;JStatsSet&quot;</td>
</tr>
<tr>
<td>SDS_PrimaryName</td>
<td>&quot;SDSPrimary&quot;</td>
</tr>
<tr>
<td>SDS_SecondaryName</td>
<td>&quot;SDSSecondary&quot;</td>
</tr>
<tr>
<td>SDS_SecondaryProxyName</td>
<td>&quot;SDSSecondaryProxy&quot;</td>
</tr>
<tr>
<td>SDS_TransactionName</td>
<td>&quot;SDSTransaction&quot;</td>
</tr>
</tbody>
</table>

For more detail, see the **JadeDynamicObject** class name property in Chapter 1 of the *JADE Encyclopaedia of Classes*. 
JadeDynamicObjectTypes Category

The global constants for the type of dynamic objects used by JADE RootSchema classes are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JStats_ArrayType</td>
<td>101</td>
</tr>
<tr>
<td>JStats_DictionaryType</td>
<td>102</td>
</tr>
<tr>
<td>JStats_JadeBytesType</td>
<td>104</td>
</tr>
<tr>
<td>JStats_SetType</td>
<td>103</td>
</tr>
<tr>
<td>SDS_PrimaryType</td>
<td>1</td>
</tr>
<tr>
<td>SDS_SecondaryProxyType</td>
<td>2</td>
</tr>
<tr>
<td>SDS_SecondaryType</td>
<td>3</td>
</tr>
<tr>
<td>SDS_TransactionType</td>
<td>4</td>
</tr>
</tbody>
</table>

For more detail, see the JadeDynamicObject class type property in Chapter 1 of the JADE Encyclopaedia of Classes.

JadeErrorCodesDatabase Category

The global constants for JADE database exception error codes that you can use in your own exception handlers, if required, are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JErr_DbDiskFull</td>
<td>3033</td>
</tr>
<tr>
<td>JErr_DbEditionOutOfDate</td>
<td>3049</td>
</tr>
<tr>
<td>JErr_DbFileExists</td>
<td>3071</td>
</tr>
<tr>
<td>JErr_DbFileNotCreated</td>
<td>3121</td>
</tr>
<tr>
<td>JErr_DbFileNotDefined</td>
<td>3120</td>
</tr>
<tr>
<td>JErr_DbFileNotFound</td>
<td>3036</td>
</tr>
<tr>
<td>JErr_DbFileOffline</td>
<td>3162</td>
</tr>
<tr>
<td>JErr_DbLockedForArchive</td>
<td>3079</td>
</tr>
<tr>
<td>JErr_DbLockedForReorg</td>
<td>3059</td>
</tr>
<tr>
<td>JErr_DbUserAbort</td>
<td>3051</td>
</tr>
<tr>
<td>JErr_FileInstantiated</td>
<td>3142</td>
</tr>
<tr>
<td>JErr_PartitionModulusRangeErr</td>
<td>3161</td>
</tr>
</tbody>
</table>

For details about exception handling, see Chapter 3 of the JADE Developer’s Reference. See also "Exception Class", in Chapter 1 of the JADE Encyclopaedia of Classes.
**JadeErrorCodesIDE Category**

The global constants for the JADE development environment error codes are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patch_History_Load_Dup_Patch</td>
<td>16007</td>
</tr>
<tr>
<td>Patch_History_Load_No_Schema</td>
<td>16006</td>
</tr>
</tbody>
</table>

**JadeErrorCodesRPS Category**

The global constants for the Relational Population Service (RPS) error codes that you can use in your own exception handlers, if required, are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JErr_DataPumpAlreadyRunning</td>
<td>3265</td>
</tr>
<tr>
<td>JErr_LegalOnRpsOnly</td>
<td>3264</td>
</tr>
<tr>
<td>JErr_NotDataPumpApp</td>
<td>3266</td>
</tr>
<tr>
<td>JErr_RpsAdminHalt</td>
<td>3262</td>
</tr>
<tr>
<td>JErr_RpsConnectionError</td>
<td>3274</td>
</tr>
<tr>
<td>JErr_RpsDuplicatedKey</td>
<td>3258</td>
</tr>
<tr>
<td>JErr_RpsExtractRequestError</td>
<td>3269</td>
</tr>
<tr>
<td>JErr_RpsMultiRowAffected</td>
<td>3260</td>
</tr>
<tr>
<td>JErr_RpsTableNameNotFound</td>
<td>3273</td>
</tr>
<tr>
<td>JErr_RpsZeroRowsAffected</td>
<td>3259</td>
</tr>
<tr>
<td>JErr_ValidOnRpsMappingOnly</td>
<td>3272</td>
</tr>
</tbody>
</table>

**JadeErrorCodesSDS Category**

The global constants for the Synchronized Database Service (SDS) error codes that you can use in your own exception handlers, if required, are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JErr_SdsIllegalOnPrimary</td>
<td>3207</td>
</tr>
<tr>
<td>JErr_SdsIllegalOnSecondary</td>
<td>3206</td>
</tr>
<tr>
<td>JErr_SdsIncompleteJournal</td>
<td>3200</td>
</tr>
<tr>
<td>JErr_SdsInvalidCommand</td>
<td>3205</td>
</tr>
<tr>
<td>JErr_SdsMaxSecondariesExceeded</td>
<td>3210</td>
</tr>
<tr>
<td>JErr_SdsNotInitialized</td>
<td>3201</td>
</tr>
<tr>
<td>JErr_SdsResponseTimeout</td>
<td>3212</td>
</tr>
</tbody>
</table>
### JadeErrorCodesWebService Category

The global constants for Web service error codes that you can use in your own exception handlers, if required, are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JADEWS_CREATE_WS_APP_FAILED</td>
<td>11082</td>
</tr>
<tr>
<td>JADEWS_DECIMAL_OVERFLOW</td>
<td>11055</td>
</tr>
<tr>
<td>JADEWS_ENUM_FAULT</td>
<td>11053</td>
</tr>
<tr>
<td>JADEWS_INVALID_REQUEST</td>
<td>11056</td>
</tr>
<tr>
<td>JADEWS_INVALID_RESPONSE</td>
<td>11051</td>
</tr>
<tr>
<td>JADEWS_LICENCES_EXCEEDED</td>
<td>11004</td>
</tr>
<tr>
<td>JADEWS_NO_WEBSERVICE_CLASS</td>
<td>11001</td>
</tr>
<tr>
<td>JADEWS_NO_WEBSERVICE_METHOD</td>
<td>11002</td>
</tr>
<tr>
<td>JADEWS_RESPONSE_TIME_EXCEEDED</td>
<td>11005</td>
</tr>
<tr>
<td>JADEWS_SERVICE_FAULT</td>
<td>11052</td>
</tr>
<tr>
<td>JADEWS_SERVICE_UNAVAILABLE</td>
<td>11008</td>
</tr>
<tr>
<td>JADEWS_SESSION_ENDED</td>
<td>11006</td>
</tr>
<tr>
<td>JADEWS_SESSION_TIMED_OUT</td>
<td>11007</td>
</tr>
<tr>
<td>JADEWS_STRING_TOO_LONG</td>
<td>11054</td>
</tr>
<tr>
<td>JADEWS_VERSION_MISMATCH</td>
<td>11009</td>
</tr>
<tr>
<td>JADEWS_WSDL_GENERATION_FAILED</td>
<td>11081</td>
</tr>
</tbody>
</table>

For more details, see Chapter 11, "Building Web Services Applications", of the JADE Developer’s Reference.

### JadeLocaleIdNumbers Category

The global constants for commonly used locale identifiers (LCIDs) are listed in the following table. You can use these values with the Application class setJadeLocale method, which changes the formatting information to suppress the regional overrides for all locales except for the LCID_SessionWithOverrides global constant.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCID_Arabic_Bahrain</td>
<td>15361</td>
</tr>
<tr>
<td>LCID_Arabic_Egypt</td>
<td>3073</td>
</tr>
</tbody>
</table>
### Global Constants Reference

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCID_Arabic_Kuwait</td>
<td>13313</td>
</tr>
<tr>
<td>LCID_Arabic_SaudiArabia</td>
<td>1025</td>
</tr>
<tr>
<td>LCID_Arabic_UAE</td>
<td>14337</td>
</tr>
<tr>
<td>LCID_Assamese_India</td>
<td>1101</td>
</tr>
<tr>
<td>LCID_Bengali_India</td>
<td>1093</td>
</tr>
<tr>
<td>LCID_Chinese_SimplifdSingapore</td>
<td>4100</td>
</tr>
<tr>
<td>LCID_Chinese_SimplifiedPRC</td>
<td>2052</td>
</tr>
<tr>
<td>LCID_Chinese_TraditionalMacao</td>
<td>5124</td>
</tr>
<tr>
<td>LCID_Chinese_TraditionalTaiwan</td>
<td>1028</td>
</tr>
<tr>
<td>LCID_Chinese_TraditionalHongKong</td>
<td>3076</td>
</tr>
<tr>
<td>LCID_Dutch_Belgium</td>
<td>2067</td>
</tr>
<tr>
<td>LCID_Dutch_Netherlands</td>
<td>1043</td>
</tr>
<tr>
<td>LCID_English_Australia</td>
<td>3081</td>
</tr>
<tr>
<td>LCID_English_Canada</td>
<td>4105</td>
</tr>
<tr>
<td>LCID_English_India</td>
<td>16393</td>
</tr>
<tr>
<td>LCID_English_Ireland</td>
<td>6153</td>
</tr>
<tr>
<td>LCID_English_Jamaica</td>
<td>8201</td>
</tr>
<tr>
<td>LCID_English_Malaysia</td>
<td>17417</td>
</tr>
<tr>
<td>LCID_English_NewZealand</td>
<td>5129</td>
</tr>
<tr>
<td>LCID_English_Singapore</td>
<td>18441</td>
</tr>
<tr>
<td>LCID_English_SouthAfrica</td>
<td>7177</td>
</tr>
<tr>
<td>LCID_English_UnitedKingdom</td>
<td>2057</td>
</tr>
<tr>
<td>LCID_English_UnitedStates</td>
<td>1033</td>
</tr>
<tr>
<td>LCID_French_Belgium</td>
<td>2060</td>
</tr>
<tr>
<td>LCID_French_Canada</td>
<td>3084</td>
</tr>
<tr>
<td>LCID_French_France</td>
<td>1036</td>
</tr>
<tr>
<td>LCID_French_Switzerland</td>
<td>4108</td>
</tr>
<tr>
<td>LCID_German_Austria</td>
<td>3079</td>
</tr>
<tr>
<td>LCID_German_Germany</td>
<td>1031</td>
</tr>
<tr>
<td>LCID_German_Switzerland</td>
<td>2055</td>
</tr>
<tr>
<td>LCID_Greek_Greece</td>
<td>1032</td>
</tr>
<tr>
<td>LCID_Gujarati_India</td>
<td>1095</td>
</tr>
<tr>
<td>LCID_Hindi_India</td>
<td>1081</td>
</tr>
<tr>
<td>LCID_Indonesian_Indonesia</td>
<td>1057</td>
</tr>
</tbody>
</table>
### Global Constants Reference

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCID_Invariant</td>
<td>127</td>
</tr>
<tr>
<td>LCID_Irish_Ireland</td>
<td>2108</td>
</tr>
<tr>
<td>LCID_Italian_Italy</td>
<td>1040</td>
</tr>
<tr>
<td>LCID_Japanese_Japan</td>
<td>1041</td>
</tr>
<tr>
<td>LCID_Kannada_India</td>
<td>1099</td>
</tr>
<tr>
<td>LCID_Konkani_India</td>
<td>1111</td>
</tr>
<tr>
<td>LCID_Korean_Korea</td>
<td>1042</td>
</tr>
<tr>
<td>LCID_Malay_Malaysia</td>
<td>1086</td>
</tr>
<tr>
<td>LCID_Malayalam_India</td>
<td>1100</td>
</tr>
<tr>
<td>LCID_Maori_NewZealand</td>
<td>1153</td>
</tr>
<tr>
<td>LCID_Marathi_India</td>
<td>1102</td>
</tr>
<tr>
<td>LCID_Oriya_India</td>
<td>1096</td>
</tr>
<tr>
<td>LCID_Polish_Poland</td>
<td>1045</td>
</tr>
<tr>
<td>LCID_Portuguese_Brazil</td>
<td>1046</td>
</tr>
<tr>
<td>LCID_Portuguese_Portugal</td>
<td>2070</td>
</tr>
<tr>
<td>LCID_Punjabi_India</td>
<td>1094</td>
</tr>
<tr>
<td>LCID_Russian_Russia</td>
<td>1049</td>
</tr>
<tr>
<td>LCID_Sanskrit_India</td>
<td>1103</td>
</tr>
<tr>
<td>LCID_SessionWithOverrides</td>
<td>1024</td>
</tr>
<tr>
<td>LCID_Spanish_Argentina</td>
<td>11274</td>
</tr>
<tr>
<td>LCID_Spanish_Chile</td>
<td>13322</td>
</tr>
<tr>
<td>LCID_Spanish_Mexico</td>
<td>2058</td>
</tr>
<tr>
<td>LCID_Spanish_Nicaragua</td>
<td>19466</td>
</tr>
<tr>
<td>LCID_Spanish_PuertoRico</td>
<td>20490</td>
</tr>
<tr>
<td>LCID_Spanish_Spain_InternatSrt</td>
<td>3082</td>
</tr>
<tr>
<td>LCID_Spanish_Spain_TradSort</td>
<td>1034</td>
</tr>
<tr>
<td>LCID_Spanish_UnitedStates</td>
<td>21514</td>
</tr>
<tr>
<td>LCID_Tamil_India</td>
<td>1097</td>
</tr>
<tr>
<td>LCID_Telugu_India</td>
<td>1098</td>
</tr>
<tr>
<td>LCID_Thai_Thailand</td>
<td>1054</td>
</tr>
<tr>
<td>LCID_Vietnamese_Vietnam</td>
<td>1066</td>
</tr>
<tr>
<td>LCID_Welsh_UnitedKingdom</td>
<td>1106</td>
</tr>
</tbody>
</table>

The LCID_Invariant global constant is an operating system-independent locale, based on English (USA). The format of the short date is **MM/dd/yyyy** and time is **HH:mm:ss**.
The **LCID_SessionWithOverrides** global constant is the Windows session locale, including user regional
overrides. It is the initial locale for the application. You can pass the enhanced locale support methods that have
an **lcid** parameter zero (0), in which case the current locale is used.

### JadeOdbc Category

The global constants for Web service error codes that you can use in your own exception handlers, if required, are
listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JErr_Attribute_Name_Conflict</td>
<td>8347</td>
</tr>
<tr>
<td>JErr_ColumnName_Cannot_Change</td>
<td>8356</td>
</tr>
<tr>
<td>JErr_Column_Not_Found</td>
<td>8349</td>
</tr>
<tr>
<td>JErr_Invalid_For_RpsMapping</td>
<td>8353</td>
</tr>
<tr>
<td>JErr_No_Jade_Type</td>
<td>8352</td>
</tr>
<tr>
<td>JErr_Not_Soft_Table</td>
<td>8351</td>
</tr>
<tr>
<td>JErr_SQL_Type_Not_Mapped</td>
<td>8348</td>
</tr>
<tr>
<td>JErr_Table_Name_Conflict</td>
<td>8345</td>
</tr>
<tr>
<td>JErr_Table_Not_Found</td>
<td>8346</td>
</tr>
</tbody>
</table>

### JadeProcessEvents Category

The global constants for JADE process events for which user notifications are sent are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process_Call_Stack_Info_Event</td>
<td>System_Base_Event + 243</td>
<td>Call stack information</td>
</tr>
<tr>
<td>Process_Local_Stats_Event</td>
<td>System_Base_Event + 240</td>
<td>Local request statistics</td>
</tr>
<tr>
<td>Process_Remote_Stats_Event</td>
<td>System_Base_Event + 241</td>
<td>Remote request statistics</td>
</tr>
<tr>
<td>Process_TDB_Analysis_Event</td>
<td>System_Base_Event + 245</td>
<td>Detailed analysis of a transient database file</td>
</tr>
<tr>
<td>Process_TDB_Info_Event</td>
<td>System_Base_Event + 244</td>
<td>File name and length of a transient database file</td>
</tr>
<tr>
<td>Process_Web_Stats_Event</td>
<td>System_Base_Event + 242</td>
<td>Web statistics information</td>
</tr>
</tbody>
</table>

### JadeProfileString Category

The global constants for JADE initialization file profiles are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProfileAllKeys</td>
<td>Returns all key strings, separated by spaces, in the initialization file section</td>
</tr>
<tr>
<td>ProfileAllSections</td>
<td>Returns all initialization file sections, separated by spaces</td>
</tr>
</tbody>
</table>
Global Constant | Description
--- | ---
ProfileRemoveKey | Removes the key string from the initialization file section
ProfileRemoveSection | Removes an entire initialization file section

For more details, see the Application class `getProfileString`, `getProfileStringAppServer`, `setProfileString`, and `setProfileStringAppServer` methods, the Process class `getProfileString` and `setProfileString` methods, and the Node class `getProfileString` and `setProfileString` methods in Chapter 1 of the JADE Encyclopaedia of Classes.

### KeyCharacterCodes Category

The global constants for printable key character codes are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Value</th>
<th>Global Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>J_key_0</td>
<td>48</td>
<td>J_key_1</td>
<td>49</td>
</tr>
<tr>
<td>J_key_2</td>
<td>50</td>
<td>J_key_3</td>
<td>51</td>
</tr>
<tr>
<td>J_key_4</td>
<td>52</td>
<td>J_key_5</td>
<td>53</td>
</tr>
<tr>
<td>J_key_6</td>
<td>54</td>
<td>J_key_7</td>
<td>55</td>
</tr>
<tr>
<td>J_key_8</td>
<td>56</td>
<td>J_key_9</td>
<td>57</td>
</tr>
<tr>
<td>J_key_A</td>
<td>65</td>
<td>J_key_Ampersand</td>
<td>38</td>
</tr>
<tr>
<td>J_key_Asterisk</td>
<td>42</td>
<td>J_key_AtSign</td>
<td>64</td>
</tr>
<tr>
<td>J_key_B</td>
<td>66</td>
<td>J_key_Back</td>
<td>8</td>
</tr>
<tr>
<td>J_key_BackSlash</td>
<td>92</td>
<td>J_key_Bar</td>
<td>8</td>
</tr>
<tr>
<td>J_key_C</td>
<td>67</td>
<td>J_key_Carat</td>
<td>94</td>
</tr>
<tr>
<td>J_key_Colon</td>
<td>58</td>
<td>J_key_Comma</td>
<td>44</td>
</tr>
<tr>
<td>J_key_Ctrl</td>
<td>17</td>
<td>J_key_CurlyLeft</td>
<td>123</td>
</tr>
<tr>
<td>J_key_CurlyRight</td>
<td>125</td>
<td>J_key_D</td>
<td>68</td>
</tr>
<tr>
<td>J_key_Delete</td>
<td>46</td>
<td>J_key_Dollar</td>
<td>36</td>
</tr>
<tr>
<td>J_key_DoubleQuote</td>
<td>34</td>
<td>J_key_DownArrow</td>
<td>40</td>
</tr>
<tr>
<td>J_key_E</td>
<td>69</td>
<td>J_key_End</td>
<td>35</td>
</tr>
<tr>
<td>J_key_Enter</td>
<td>13</td>
<td>J_key_Equal</td>
<td>61</td>
</tr>
<tr>
<td>J_key_Escape</td>
<td>27</td>
<td>J_key_Exclamation</td>
<td>33</td>
</tr>
<tr>
<td>J_key_F</td>
<td>70</td>
<td>J_key_F1</td>
<td>112</td>
</tr>
<tr>
<td>J_key_F10</td>
<td>121</td>
<td>J_key_F11</td>
<td>122</td>
</tr>
<tr>
<td>J_key_F12</td>
<td>123</td>
<td>J_key_F2</td>
<td>113</td>
</tr>
<tr>
<td>J_key_F3</td>
<td>114</td>
<td>J_key_F4</td>
<td>115</td>
</tr>
<tr>
<td>J_key_F5</td>
<td>116</td>
<td>J_key_F6</td>
<td>117</td>
</tr>
<tr>
<td>J_key_F7</td>
<td>118</td>
<td>J_key_F8</td>
<td>119</td>
</tr>
<tr>
<td>Global Constant</td>
<td>Value</td>
<td>Global Constant</td>
<td>Value</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------</td>
<td>-----------------------</td>
<td>-------</td>
</tr>
<tr>
<td>J_key_F9</td>
<td>120</td>
<td>J_key_G</td>
<td>71</td>
</tr>
<tr>
<td>J_key_GreaterThan</td>
<td>62</td>
<td>J_key_H</td>
<td>72</td>
</tr>
<tr>
<td>J_key_Hash</td>
<td>35</td>
<td>J_key_Home</td>
<td>36</td>
</tr>
<tr>
<td>J_key_Hyphen</td>
<td>45</td>
<td>J_key_I</td>
<td>73</td>
</tr>
<tr>
<td>J_key_Insert</td>
<td>45</td>
<td>J_key_J</td>
<td>74</td>
</tr>
<tr>
<td>J_key_K</td>
<td>75</td>
<td>J_key_L</td>
<td>76</td>
</tr>
<tr>
<td>J_key_LeftArrow</td>
<td>37</td>
<td>J_key_LeftBracket</td>
<td>91</td>
</tr>
<tr>
<td>J_key_LeftParenthesis</td>
<td>40</td>
<td>J_key_LeftQuote</td>
<td>96</td>
</tr>
<tr>
<td>J_key_LessThan</td>
<td>60</td>
<td>J_key_Linefeed</td>
<td>10</td>
</tr>
<tr>
<td>J_key_M</td>
<td>77</td>
<td>J_key_N</td>
<td>78</td>
</tr>
<tr>
<td>J_key_NumPadMinus</td>
<td>109</td>
<td>J_key_NumPadMultiply</td>
<td>106</td>
</tr>
<tr>
<td>J_key_NumPadPlus</td>
<td>107</td>
<td>J_key_O</td>
<td>79</td>
</tr>
<tr>
<td>J_key_P</td>
<td>80</td>
<td>J_key_PageDown</td>
<td>34</td>
</tr>
<tr>
<td>J_key_PageUp</td>
<td>33</td>
<td>J_key_Percent</td>
<td>37</td>
</tr>
<tr>
<td>J_key_Plus</td>
<td>43</td>
<td>J_key_Q</td>
<td>81</td>
</tr>
<tr>
<td>J_key_Question</td>
<td>63</td>
<td>J_key_R</td>
<td>82</td>
</tr>
<tr>
<td>J_key_Return</td>
<td>13</td>
<td>J_key_RightArrow</td>
<td>39</td>
</tr>
<tr>
<td>J_key_RightBracket</td>
<td>93</td>
<td>J_key_RightParenthesis</td>
<td>41</td>
</tr>
<tr>
<td>J_key_S</td>
<td>83</td>
<td>J_key_SemiColon</td>
<td>59</td>
</tr>
<tr>
<td>J_key_Shift</td>
<td>16</td>
<td>J_key_SingleQuote</td>
<td>39</td>
</tr>
<tr>
<td>J_key_Slash</td>
<td>47</td>
<td>J_key_Space</td>
<td>32</td>
</tr>
<tr>
<td>J_key_Stop</td>
<td>46</td>
<td>J_key_T</td>
<td>84</td>
</tr>
<tr>
<td>J_key_Tab</td>
<td>9</td>
<td>J_key_Tilde</td>
<td>126</td>
</tr>
<tr>
<td>J_key_U</td>
<td>85</td>
<td>J_key_UnderScore</td>
<td>95</td>
</tr>
<tr>
<td>J_key_UpArrow</td>
<td>38</td>
<td>J_key_V</td>
<td>86</td>
</tr>
<tr>
<td>J_key_W</td>
<td>87</td>
<td>J_key_X</td>
<td>88</td>
</tr>
<tr>
<td>J_key_Y</td>
<td>89</td>
<td>J_key_Z</td>
<td>90</td>
</tr>
<tr>
<td>J_key_a</td>
<td>97</td>
<td>J_key_b</td>
<td>98</td>
</tr>
<tr>
<td>J_key_c</td>
<td>99</td>
<td>J_key_d</td>
<td>100</td>
</tr>
<tr>
<td>J_key_e</td>
<td>101</td>
<td>J_key_f</td>
<td>102</td>
</tr>
<tr>
<td>J_key_g</td>
<td>103</td>
<td>J_key_h</td>
<td>104</td>
</tr>
<tr>
<td>J_key_i</td>
<td>105</td>
<td>J_key_j</td>
<td>106</td>
</tr>
<tr>
<td>J_key_k</td>
<td>107</td>
<td>J_key_l</td>
<td>108</td>
</tr>
<tr>
<td>J_key_m</td>
<td>109</td>
<td>J_key_n</td>
<td>110</td>
</tr>
</tbody>
</table>
The global constants for lock durations are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistent_Duration</td>
<td>2</td>
<td>Reserved for future use (not yet implemented).</td>
</tr>
<tr>
<td>Session_Duration</td>
<td>1</td>
<td>Automatically unlocks the object at the end of the current session (that is, the current thread or process) if no manual unlocks are issued. In persistent transaction state, all unlock requests for persistent objects are ignored. Similarly, in transient transaction state, all unlock requests for shared transient objects are ignored. A session lock is therefore not released if the unlock request is made while in transaction state. To release a session lock, the unlock request must be made while not in transaction state.</td>
</tr>
<tr>
<td>Transaction_Duration</td>
<td>0</td>
<td>Automatically unlocks the object at the end of transaction time. If a manual unlock is issued, this unlocks the object only if you are not in transaction or load state.</td>
</tr>
</tbody>
</table>

The global constants for lock timeouts are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LockTimeout_Immediate</td>
<td>-1</td>
<td>Lock request times out immediately</td>
</tr>
<tr>
<td>LockTimeout_Infinite</td>
<td>Max_Integer (#7FFFFFFF)</td>
<td>Lock request times out only after the number of milliseconds indicated by the Max_Integer value is reached</td>
</tr>
<tr>
<td>LockTimeout_Server_DEFINED</td>
<td>0</td>
<td>Uses the server-defined default</td>
</tr>
</tbody>
</table>
Locks Category

The global constants for locks are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exclusive_Lock</td>
<td>3</td>
<td>No other process can lock the same object.</td>
</tr>
<tr>
<td>Get_Lock</td>
<td>0</td>
<td>Not valid for lock requests. This lock type indicates a process is waiting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to acquire a lock that will cause all other lock requests for the object to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>be queued (for example, when upgrading a lock from update to exclusive).</td>
</tr>
<tr>
<td>Reserve_Lock</td>
<td>2</td>
<td>When you place a reserve lock on an object, other processes attempting to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>acquire an exclusive lock or reserve lock on that same object wait until</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the reserve lock is relinquished, but those attempting to acquire a shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lock succeed.</td>
</tr>
<tr>
<td>Share_Lock</td>
<td>1</td>
<td>When you place a shared lock on an object, other processes attempting to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>update the object or explicitly acquire an exclusive lock wait until the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lock is released but can acquire a shared lock or a reserve lock.</td>
</tr>
<tr>
<td>Update_Lock</td>
<td>4</td>
<td>Placing an update lock allows you to update the object, while still</td>
</tr>
<tr>
<td></td>
<td></td>
<td>allowing other processes to acquire shared locks to view the most recently</td>
</tr>
<tr>
<td></td>
<td></td>
<td>committed edition.</td>
</tr>
</tbody>
</table>

MessageBox Category

The global constants for message boxes are listed in the following table. (For more details, see the Application class msgBox method in Chapter 1 of the JADE Encyclopaedia of Classes.)

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MsgBox_Abort_Retry_Ignore</td>
<td>2</td>
<td>Displays the Abort, Retry, and Ignore buttons</td>
</tr>
<tr>
<td>MsgBox_App_Modal</td>
<td>0</td>
<td>User must respond to the message box before continuing work</td>
</tr>
<tr>
<td>MsgBox_Default_First</td>
<td>0</td>
<td>First button is the default</td>
</tr>
<tr>
<td>MsgBox_Default_Second</td>
<td>256</td>
<td>Second button is the default</td>
</tr>
<tr>
<td>MsgBox_Default_Third</td>
<td>512</td>
<td>Third button is the default</td>
</tr>
<tr>
<td>MsgBox_Exclamation_Mark.Icon</td>
<td>48</td>
<td>Displays the Exclamation Mark icon</td>
</tr>
<tr>
<td>MsgBox_Information.Icon</td>
<td>64</td>
<td>Displays the Information icon</td>
</tr>
<tr>
<td>MsgBox_OK_Cancel</td>
<td>1</td>
<td>Displays the OK and Cancel buttons</td>
</tr>
<tr>
<td>MsgBox_OK_Only</td>
<td>0</td>
<td>Displays only the OK button</td>
</tr>
<tr>
<td>MsgBox_Question_Mark.Icon</td>
<td>32</td>
<td>Displays the Question Mark icon</td>
</tr>
<tr>
<td>MsgBox_Retry_Cancel</td>
<td>5</td>
<td>Displays the Retry and Cancel buttons</td>
</tr>
<tr>
<td>MsgBox_Return_Abort</td>
<td>3</td>
<td>Returned when the Abort button has been selected</td>
</tr>
<tr>
<td>MsgBox_Return_Cancel</td>
<td>2</td>
<td>Returned when the Cancel button or the ESC key has been selected</td>
</tr>
</tbody>
</table>
### NotificationResponses Category

The global constants for event notification responses are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MsgBox_Return_Ignore</td>
<td>5</td>
<td>Returned when the Ignore button has been selected</td>
</tr>
<tr>
<td>MsgBox_Return_No</td>
<td>7</td>
<td>Returned when the No button has been selected</td>
</tr>
<tr>
<td>MsgBox_Return_OK</td>
<td>1</td>
<td>Returned when the OK button has been selected</td>
</tr>
<tr>
<td>MsgBox_Return_Retry</td>
<td>4</td>
<td>Returned when the Retry button has been selected</td>
</tr>
<tr>
<td>MsgBox_Return_Yes</td>
<td>6</td>
<td>Returned when the Yes button has been selected</td>
</tr>
<tr>
<td>MsgBox_Stop_Icon</td>
<td>16</td>
<td>Displays the Stop icon</td>
</tr>
<tr>
<td>MsgBox_System Modal</td>
<td>4096</td>
<td>All applications are suspended until the user responds to the message box</td>
</tr>
<tr>
<td>MsgBox_Yes_No</td>
<td>4</td>
<td>Displays Yes and No buttons</td>
</tr>
<tr>
<td>MsgBox_Yes_No_Cancel</td>
<td>3</td>
<td>Displays Yes, No, and Cancel buttons</td>
</tr>
</tbody>
</table>

### ObjectVolatility Category

The global constants for the volatility state of persistent objects are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Object is…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volatility_Frozen</td>
<td>#04</td>
<td>Frozen (that is, it is not updated)</td>
</tr>
<tr>
<td>Volatility_Stable</td>
<td>#08</td>
<td>Stable (that is, it is updated infrequently)</td>
</tr>
<tr>
<td>Volatility_Volatile</td>
<td>#00</td>
<td>Volatile (that is, it is updated often)</td>
</tr>
</tbody>
</table>

For details, see "Cache Concurrency", in Chapter 6 of the JADE Developer’s Reference.
PossibleTransientLeaks Category

The global constant that enables you to mark lines of code for exclusion from the transient leaks analysis report is listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExcludeFromTransientLeakReport</td>
<td>Marks the line of code for exclusion from the transient leaks analysis report when specified in a comment on the same line</td>
</tr>
</tbody>
</table>

For details about possible transient leak analysis, see "Locating Possible Transient ObjectLeaks", in Chapter 4 of the JADE Development Environment User's Guide.

Printer Category

The printer global constants are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print_10X11</td>
<td>45</td>
<td>10 x 11 inches</td>
</tr>
<tr>
<td>Print_10X14</td>
<td>16</td>
<td>10 x 14 inches</td>
</tr>
<tr>
<td>Print_11X17</td>
<td>17</td>
<td>11 x 17 inches</td>
</tr>
<tr>
<td>Print_15X11</td>
<td>46</td>
<td>15 x 11 inches</td>
</tr>
<tr>
<td>Print_9X11</td>
<td>44</td>
<td>9 x 11 inches</td>
</tr>
<tr>
<td>Print_A2</td>
<td>66</td>
<td>A2 420 x 594 mm</td>
</tr>
<tr>
<td>Print_A3</td>
<td>8</td>
<td>A3 297 x 420 mm</td>
</tr>
<tr>
<td>Print_A3_Extra</td>
<td>63</td>
<td>A3 Extra 322 x 445 mm</td>
</tr>
<tr>
<td>Print_A3_Extra_Transverse</td>
<td>68</td>
<td>A3 Extra Transverse</td>
</tr>
<tr>
<td>Print_A3_Transverse</td>
<td>67</td>
<td>A3 Transverse 297 x 420 mm</td>
</tr>
<tr>
<td>Print_A4</td>
<td>9</td>
<td>A4 210 x 297 mm</td>
</tr>
<tr>
<td>Print_A4Small</td>
<td>10</td>
<td>A4 Small 210 x 297 mm</td>
</tr>
<tr>
<td>Print_A4_EXTRA</td>
<td>53</td>
<td>A4 Extra 9.27 x 12.69 inches</td>
</tr>
<tr>
<td>Print_A4_PLUS</td>
<td>60</td>
<td>A4 Plus 210 x 330 mm</td>
</tr>
<tr>
<td>Print_A4_TRANSVERSE</td>
<td>55</td>
<td>A4 Transverse 210 x 297 mm</td>
</tr>
<tr>
<td>Print_A5</td>
<td>11</td>
<td>A5 148 x 210 mm</td>
</tr>
<tr>
<td>Print_A5_EXTRA</td>
<td>64</td>
<td>A5 Extra 174 x 235 mm</td>
</tr>
<tr>
<td>Print_A5_TRANSVERSE</td>
<td>61</td>
<td>A5 Transverse 148 x 210 mm</td>
</tr>
<tr>
<td>Print_A_PLUS</td>
<td>57</td>
<td>Super A - A4 227 x 356 mm</td>
</tr>
<tr>
<td>Print_B4</td>
<td>12</td>
<td>B4 250 x 354 mm</td>
</tr>
<tr>
<td>Print_B5</td>
<td>13</td>
<td>B5 182 x 257 mm</td>
</tr>
<tr>
<td>Global Constant</td>
<td>Integer Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Print_B5_Extra</td>
<td>65</td>
<td>B5 (ISO) Extra 201 x 276 mm</td>
</tr>
<tr>
<td>Print_B5_Transverse</td>
<td>62</td>
<td>B5 (JIS) Transverse 182 x 257 mm</td>
</tr>
<tr>
<td>Print_B_Plus</td>
<td>58</td>
<td>Super B – A3 305 x 487 mm</td>
</tr>
<tr>
<td>Print_CSheet</td>
<td>24</td>
<td>C size sheet</td>
</tr>
<tr>
<td>Print_Cancelled</td>
<td>15015</td>
<td>Print progress dialog <strong>Cancel</strong> button was clicked</td>
</tr>
<tr>
<td>Print_Collate_Ignored</td>
<td>15030</td>
<td>Printing started, so change to the <strong>collate</strong> property ignored</td>
</tr>
<tr>
<td>Print_Copies_Ignored</td>
<td>15010</td>
<td>Printing started, so change of copies ignored</td>
</tr>
<tr>
<td>Print_Currently_Open</td>
<td>15013</td>
<td>Printer is currently open</td>
</tr>
<tr>
<td>Print_Custom_Paper</td>
<td>256</td>
<td>Customized paper size</td>
</tr>
<tr>
<td>Print_DSheet</td>
<td>25</td>
<td>D size sheet</td>
</tr>
<tr>
<td>Print_DocumentType_Invalid</td>
<td>15032</td>
<td>You changed <strong>printer.documentType</strong> to <strong>Print_Custom_Paper</strong> instead of calling the <strong>printer.setCustomPaperSize</strong> method</td>
</tr>
<tr>
<td>Print_Duplex_Ignored</td>
<td>15029</td>
<td>Printing started, so change to the <strong>duplex</strong> property ignored</td>
</tr>
<tr>
<td>Print_Duplex_Invalid</td>
<td>15028</td>
<td>Value of the <strong>duplex</strong> property is invalid</td>
</tr>
<tr>
<td>Print_ESheet</td>
<td>26</td>
<td>E size sheet</td>
</tr>
<tr>
<td>Print_Env_10</td>
<td>20</td>
<td>Envelope #10 4.125 x 9.5 inches</td>
</tr>
<tr>
<td>Print_Env_11</td>
<td>21</td>
<td>Envelope #11 4.5 x 10.375 inches</td>
</tr>
<tr>
<td>Print_Env_12</td>
<td>22</td>
<td>Envelope #12 4.75 x 11 inches</td>
</tr>
<tr>
<td>Print_Env_14</td>
<td>23</td>
<td>Envelope #14 5 x 11.5 inches</td>
</tr>
<tr>
<td>Print_Env_9</td>
<td>19</td>
<td>Envelope #9 3.875 x 8.875 inches</td>
</tr>
<tr>
<td>Print_Env_B4</td>
<td>33</td>
<td>Envelope B4 250 x 353 mm</td>
</tr>
<tr>
<td>Print_Env_B5</td>
<td>34</td>
<td>Envelope B5 176 x 250 mm</td>
</tr>
<tr>
<td>Print_Env_B6</td>
<td>35</td>
<td>Envelope B6 176 x 125 mm</td>
</tr>
<tr>
<td>Print_Env_C3</td>
<td>29</td>
<td>Envelope C3 324 x 458 mm</td>
</tr>
<tr>
<td>Print_Env_C4</td>
<td>30</td>
<td>Envelope C4 229 x 324 mm</td>
</tr>
<tr>
<td>Print_Env_C5</td>
<td>28</td>
<td>Envelope C5 162 x 229 mm</td>
</tr>
<tr>
<td>Print_Env_C6</td>
<td>31</td>
<td>Envelope C6 114 x 162 mm</td>
</tr>
<tr>
<td>Print_Env_C65</td>
<td>32</td>
<td>Envelope C65 114 x 229 mm</td>
</tr>
<tr>
<td>Print_Env_DL</td>
<td>27</td>
<td>Envelope DL 110 x 220 mm</td>
</tr>
<tr>
<td>Print_Env_Invite</td>
<td>47</td>
<td>Envelope Invite 220 x 220 mm</td>
</tr>
<tr>
<td>Print_Env_Italy</td>
<td>36</td>
<td>Envelope 110 x 230 mm</td>
</tr>
<tr>
<td>Print_Env_Monarch</td>
<td>37</td>
<td>Envelope Monarch 3.875 x 7.5 inches</td>
</tr>
<tr>
<td>Print_Env_Personal</td>
<td>38</td>
<td>6¾ Envelope 3.625 x 6.5 inches</td>
</tr>
</tbody>
</table>
## Global Constants Reference

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print_Executive</td>
<td>7</td>
<td>Executive 7⅞ x 10⅜ inches</td>
</tr>
<tr>
<td>Print_Failed_To_Obtain_Printer</td>
<td>15014</td>
<td>Task could not obtain printer</td>
</tr>
<tr>
<td>Print_Fanfold_Lgl_German</td>
<td>41</td>
<td>German Legal Fanfold 8⅜ x 13 inches</td>
</tr>
<tr>
<td>Print_Fanfold_Std_German</td>
<td>40</td>
<td>German Std Fanfold 8⅛ x 12 inches</td>
</tr>
<tr>
<td>Print_Fanfold_US</td>
<td>39</td>
<td>US Std Fanfold 14.875 x 11 inches</td>
</tr>
<tr>
<td>Print_Folio</td>
<td>14</td>
<td>Folio 8⅛ x 13 inches</td>
</tr>
<tr>
<td>Print_Frame_Too_Large</td>
<td>15007</td>
<td>Frame larger than page depth</td>
</tr>
<tr>
<td>Print_Header_Footer_Too_Large</td>
<td>15006</td>
<td>Header and footer larger than page depth</td>
</tr>
<tr>
<td>Print_In_Preview</td>
<td>15031</td>
<td>Print object in use in print preview and cannot be reused</td>
</tr>
<tr>
<td>Print_ISO_B4</td>
<td>42</td>
<td>B4 (ISO) 250 x 353 mm</td>
</tr>
<tr>
<td>Print_Invalid_Control</td>
<td>15001</td>
<td>Attempt to print control that is not a frame</td>
</tr>
<tr>
<td>Print_Invalid_Position</td>
<td>15024</td>
<td>Attempt to set a print position that is outside the valid range</td>
</tr>
<tr>
<td>Print_Japanese_PostCard</td>
<td>43</td>
<td>Japanese Postcard 100 x 148 mm</td>
</tr>
<tr>
<td>Print_Landscape</td>
<td>2</td>
<td>Horizontal page orientation</td>
</tr>
<tr>
<td>Print_Ledger</td>
<td>4</td>
<td>Ledger 17 x 11 inches</td>
</tr>
<tr>
<td>Print_Legal</td>
<td>5</td>
<td>Legal 8½ x 14 inches</td>
</tr>
<tr>
<td>Print_Legal_Extra</td>
<td>51</td>
<td>Legal Extra 9.275 x 15 inches</td>
</tr>
<tr>
<td>Print_Letter</td>
<td>1</td>
<td>Letter 8½ x 11 inches</td>
</tr>
<tr>
<td>Print_LetterSmall</td>
<td>2</td>
<td>Letter Small 8½ x 11 inches</td>
</tr>
<tr>
<td>Print_Letter_Extra</td>
<td>50</td>
<td>Letter Extra 9.275 x 12 inches</td>
</tr>
<tr>
<td>Print_Letter_Extra_Transverse</td>
<td>56</td>
<td>Letter Extra Transverse 9.275 x 12 inches</td>
</tr>
<tr>
<td>Print_Letter_Plus</td>
<td>59</td>
<td>Letter Plus 8.5 x 12.69 inches</td>
</tr>
<tr>
<td>Print_Letter_Transverse</td>
<td>54</td>
<td>Letter Transverse 8.275 x 11 inches</td>
</tr>
<tr>
<td>Print_Metafile_Playback_Error</td>
<td>15033</td>
<td>Internal error occurred when attempting to play back a print metafile</td>
</tr>
<tr>
<td>Print_NewPage_Failed</td>
<td>15002</td>
<td>New page on printer failed</td>
</tr>
<tr>
<td>Print_NoDefault_Printer</td>
<td>15021</td>
<td>Your workstation has no default printer set up</td>
</tr>
<tr>
<td>Print_Not_Available</td>
<td>15017</td>
<td>Printer does not match available printers</td>
</tr>
<tr>
<td>Print_Orientation_Invalid</td>
<td>15011</td>
<td>Orientation must be Portrait (1) or Landscape (2)</td>
</tr>
<tr>
<td>Print_PaperSource_Invalid</td>
<td>15027</td>
<td>Value of the paperSource property is invalid</td>
</tr>
<tr>
<td>Print_Portion</td>
<td>1</td>
<td>Vertical page orientation</td>
</tr>
<tr>
<td>Print_Preview_Ignored</td>
<td>15008</td>
<td>Printing started, so change of print preview ignored</td>
</tr>
<tr>
<td>Print_PrintReport_Ignored</td>
<td>15022</td>
<td>Printing started, so change of print report ignored</td>
</tr>
</tbody>
</table>
Appendix A  Global Constants Reference

Global Constant  | Integer Value | Description
-----------------|---------------|-----------------------------------------------
Print_Printer_Ignored  | 15023  | You attempted to change the printer in use after printing began or before any printing has occurred (the printer must be closed before commencing the new output on a different printer)
Print_Printer_Not_Open  | 15003  | Printer not open
Print_Printer_Open_Failed  | 15005  | Open of printer failed
Print_Quarto  | 15  | Quarto 215 x 275 mm
Print.Restricted  | 15019  | Statement 5½ x 8½ inches
Print.Statement  | 6  | Statement 5½ x 8½ inches
Print.Stopped  | 15016  | Print progress dialog Stop button was clicked
Print.Successful  | 0  | The print was successful
Print_Tabloid  | 3  | Tabloid 11 x 17 inches
Print_Tabloid.Extra  | 52  | Tabloid Extra 11.69 x 18 inches
Print_TextOut_Error  | 15004  | Text output to printer failed
Print.UnformattedFailed  | 15025  | Printing of unformatted text failed; that is, the printUnformatted method request failed

RPSTransitionHaltCode Category

The global constants that enable you to determine whether the Relational Population Service (RPS) table alter script will be loaded automatically or whether manual action is required from the RDB administrator when a schema instantiation is replayed by an RPS node and the data pump application and database tracking are halted to achieve a schema transition.

When the event RPS_SchemaTransition (event type 220) is caused on the system instance, the userInfo parameter passed to the userNotification callback method contains one of the global constants listed in the following table. (The RPS_SchemaTransition event is represented by a global constant in the SDSEventTypes category.)

Global Constant  | Integer Value | Description
-----------------|---------------|-----------------------------------------------
RPS_HaltAutoScript  | 1  | An automatic initiate alter script was generated (will be automatically loaded by the data pump application if configured to automatically restart)
RPS_HaltManualScript  | 2  | A manual alter script was generated (requires administration user intervention to apply changes to RDB before tracking can be resumed)
RPS_HaltMappingDeleted  | 3  | The RPS mapping was deleted on the primary database, rendering the RPS node and associated RDB defunct
RPS_HaltNoScript  | 0  | Changes do not affect RDB, so no script was generated
The global constants for the Synchronized Database Service (SDS) connection state are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_Connected</td>
<td>2</td>
</tr>
<tr>
<td>SDS_Connecting</td>
<td>3</td>
</tr>
<tr>
<td>SDS_ConnectionFailed</td>
<td>4</td>
</tr>
<tr>
<td>SDS_ConnectionStateUndefined</td>
<td>0</td>
</tr>
<tr>
<td>SDS_Disconnected</td>
<td>1</td>
</tr>
</tbody>
</table>

These are used in return values or dynamic object attribute values by the `JadeDatabaseAdmin` class `sdsGetMyServerInfo` method.

The global constants for the Synchronized Database Service (SDS) database roles are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_RolePrimary</td>
<td>1</td>
</tr>
<tr>
<td>SDS_RoleSecondary</td>
<td>2</td>
</tr>
<tr>
<td>SDS_RoleUndefined (returned when the method is invoked on a non-SDS-capable or non-RPS-capable system)</td>
<td>0</td>
</tr>
<tr>
<td>SDS_SubroleNative</td>
<td>1</td>
</tr>
<tr>
<td>SDS_SubroleRelational</td>
<td>2</td>
</tr>
</tbody>
</table>

These are used in return values or dynamic object attribute values by the `JadeDatabaseAdmin` class `sdsGetMyServerInfo` or `sdsGetDatabaseRole` method.

The global constants for the Synchronized Database Service (SDS) event types are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPS_SchemaTransition</td>
<td>220</td>
</tr>
<tr>
<td>SDS_ConnectionStateChange</td>
<td>17386</td>
</tr>
<tr>
<td>SDS_JournalTransferStopped</td>
<td>17385</td>
</tr>
<tr>
<td>SDS_RoleChangeEvent</td>
<td>22</td>
</tr>
<tr>
<td>SDS_RoleChangeProgress</td>
<td>17387</td>
</tr>
</tbody>
</table>
### Global Constants Reference

**SDS Reorganization State Category**

The global constants for the Synchronized Database Service (SDS) reorganization state are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_ReorgStateNotReorging</td>
<td>1</td>
</tr>
<tr>
<td>SDS_ReorgStateOfflinePhase</td>
<td>5</td>
</tr>
<tr>
<td>SDS_ReorgStateReorgingFiles</td>
<td>4</td>
</tr>
<tr>
<td>SDS_ReorgStateRestarting</td>
<td>6</td>
</tr>
<tr>
<td>SDS_ReorgStateSeekingApproval</td>
<td>2</td>
</tr>
<tr>
<td>SDS_ReorgStateStarting</td>
<td>3</td>
</tr>
</tbody>
</table>

These are used in return values or dynamic object attribute values by the `JadeDatabaseAdmin` class `sdsGetMyServerInfo` method.

**SDS Secondary State Category**

The global constants for the Synchronized Database Service (SDS) secondary state are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_BlockWrite</td>
<td>2</td>
</tr>
<tr>
<td>SDS_JournalSwitch</td>
<td>1</td>
</tr>
<tr>
<td>SDS_StateCatchingUp</td>
<td>1</td>
</tr>
<tr>
<td>SDS_StateDisconnected</td>
<td>0</td>
</tr>
<tr>
<td>SDS_StateReorging</td>
<td>5</td>
</tr>
<tr>
<td>SDS_StateSynchronized</td>
<td>2</td>
</tr>
<tr>
<td>SDS_StateTrackingHalted</td>
<td>4</td>
</tr>
<tr>
<td>SDS_StateTransferHalted</td>
<td>3</td>
</tr>
</tbody>
</table>

These are used in return values or dynamic object attribute values by the `JadeDatabaseAdmin` class `sdsGetMyServerInfo` method.
These are used in return values or dynamic object attribute values by the *JadeDatabaseAdmin* class *sdsGetMyServerInfo* method.

### SDSStopTrackingCodes Category

The global constants for the Synchronized Database Service (SDS) stop tracking are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_AuditStopTrackingAll</td>
<td>1</td>
</tr>
<tr>
<td>SDS_AuditStopTrackingNative</td>
<td>2</td>
</tr>
<tr>
<td>SDS_AuditStopTrackingRdb</td>
<td>3</td>
</tr>
<tr>
<td>SDS_ReasonAdminAudited</td>
<td>1</td>
</tr>
<tr>
<td>SDS_ReasonAdminDirect</td>
<td>2</td>
</tr>
<tr>
<td>SDS_ReasonAutoUpgradeMismatch</td>
<td>6</td>
</tr>
<tr>
<td>SDS_ReasonDeltaModeEntered</td>
<td>12</td>
</tr>
<tr>
<td>SDS_ReasonEnablingDbCrypt</td>
<td>13</td>
</tr>
<tr>
<td>SDS_ReasonErrorHalt</td>
<td>8</td>
</tr>
<tr>
<td>SDS_ReasonRestart</td>
<td>10</td>
</tr>
<tr>
<td>SDS_ReasonRpsAdminHalt</td>
<td>4</td>
</tr>
<tr>
<td>SDS_ReasonRpsReorgHalt</td>
<td>9</td>
</tr>
<tr>
<td>SDS_ReasonRpsRestart</td>
<td>11</td>
</tr>
<tr>
<td>SDS_ReasonRpsSnapshot</td>
<td>3</td>
</tr>
<tr>
<td>SDS_ReasonTakeover</td>
<td>7</td>
</tr>
<tr>
<td>SDS_ReasonTransition</td>
<td>5</td>
</tr>
</tbody>
</table>

These are used in return values or dynamic object attribute values by the *JadeDatabaseAdmin* class *getReasonTrackingStoppedString*, *sdsAuditStopTracking*, *sdsGetMyServerInfo*, and *sdsGetSecondaryInfo* methods.

### SDSTakeoverState Category

The global constants for the Synchronized Database Service (SDS) takeover state are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_HostileTakeoverInitiated</td>
<td>4</td>
</tr>
<tr>
<td>SDS_PrimaryRoleActive</td>
<td>1</td>
</tr>
<tr>
<td>SDS_PrimaryRoleRelinquished</td>
<td>11</td>
</tr>
<tr>
<td>SDS_RelinquishPrimaryRole</td>
<td>5</td>
</tr>
<tr>
<td>SDS_RelinquishSecondaryRole</td>
<td>6</td>
</tr>
</tbody>
</table>
EncycloPrim - 7.1.08

Global Constants Reference

SDSTransactionStates Category

The global constants for the Synchronized Database Service (SDS) transaction states are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_TranDeferred</td>
<td>3</td>
</tr>
<tr>
<td>SDS_TranInDoubt</td>
<td>8</td>
</tr>
<tr>
<td>SDS_TranInterrupted</td>
<td>2</td>
</tr>
<tr>
<td>SDS_TranNormal</td>
<td>1</td>
</tr>
<tr>
<td>SDS_TranPrepareToCommit</td>
<td>6</td>
</tr>
<tr>
<td>SDS_TranReadyToAbort</td>
<td>7</td>
</tr>
<tr>
<td>SDS_TranReadyToCommit</td>
<td>5</td>
</tr>
<tr>
<td>SDS_TranWaitingAuditCommit</td>
<td>4</td>
</tr>
</tbody>
</table>

These values are used in return values or dynamic object attribute values by the **JadeDatabaseAdmin** class **sdsGetMyServerInfo**, **sdsGetTransactions**, or **sdsGetTransactionsAt** method.

SQL Category

The global constants for the Structured Query Language (SQL) are listed in the following table. The SQL global constants are for internal use only.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_COLLECTION</td>
<td>3</td>
</tr>
</tbody>
</table>

These values are used as the values contained in the **userInfo** parameter for a role change progress event notification. For details, see "Detecting SDS Role Changes", in Chapter 10 of the **JADE Developer’s Reference**. See also the **sdsInitiateTakeover** method of the **JadeDatabaseAdmin** class.
<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_COLLECTION_METHOD</td>
<td>7</td>
</tr>
<tr>
<td>SQL_EXPPLICIT_INVERSE</td>
<td>4</td>
</tr>
<tr>
<td>SQL_IMPLICIT_INVERSE</td>
<td>5</td>
</tr>
<tr>
<td>SQL_INVALID</td>
<td>0</td>
</tr>
<tr>
<td>SQL_METHOD</td>
<td>2</td>
</tr>
<tr>
<td>SQL_PROPERTY</td>
<td>1</td>
</tr>
<tr>
<td>SQL_TYPE_BIGINT</td>
<td>-5</td>
</tr>
<tr>
<td>SQL_TYPE_BINARY</td>
<td>-2</td>
</tr>
<tr>
<td>SQL_TYPE_BIT</td>
<td>-7</td>
</tr>
<tr>
<td>SQL_TYPE_CHAR</td>
<td>1</td>
</tr>
<tr>
<td>SQL_TYPE_DATE</td>
<td>9</td>
</tr>
<tr>
<td>SQL_TYPE_DATE_VERSION3</td>
<td>91</td>
</tr>
<tr>
<td>SQL_TYPE_DECIMAL</td>
<td>3</td>
</tr>
<tr>
<td>SQL_TYPE_DOUBLE</td>
<td>8</td>
</tr>
<tr>
<td>SQL_TYPE_FLOAT</td>
<td>6</td>
</tr>
<tr>
<td>SQL_TYPE_INTEGER</td>
<td>4</td>
</tr>
<tr>
<td>SQL_TYPE_INTERVAL_DAY_TO_SEC</td>
<td>110</td>
</tr>
<tr>
<td>SQL_TYPE_LONGVARBINARY</td>
<td>-4</td>
</tr>
<tr>
<td>SQL_TYPE_LONGVARCHAR</td>
<td>-1</td>
</tr>
<tr>
<td>SQL_TYPE_NULL</td>
<td>0</td>
</tr>
<tr>
<td>SQL_TYPE_NUMERIC</td>
<td>2</td>
</tr>
<tr>
<td>SQL_TYPE_OID</td>
<td>-50</td>
</tr>
<tr>
<td>SQL_TYPE_REAL</td>
<td>7</td>
</tr>
<tr>
<td>SQL_TYPE_SMALLINT</td>
<td>5</td>
</tr>
<tr>
<td>SQL_TYPE_TIME</td>
<td>10</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>11</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP_VERSION3</td>
<td>93</td>
</tr>
<tr>
<td>SQL_TYPE_TIME_VERSION3</td>
<td>92</td>
</tr>
<tr>
<td>SQL_TYPE_TINYINT</td>
<td>-6</td>
</tr>
<tr>
<td>SQL_TYPE_VARBINARY</td>
<td>-3</td>
</tr>
<tr>
<td>SQL_TYPE_VARCHAR</td>
<td>12</td>
</tr>
<tr>
<td>SQL_TYPE_WCHAR</td>
<td>-8</td>
</tr>
<tr>
<td>SQL_TYPE_WLONGVARCHAR</td>
<td>-10</td>
</tr>
<tr>
<td>SQL_TYPE_WVARCHAR</td>
<td>-9</td>
</tr>
<tr>
<td>SQL_XKEYDICT</td>
<td>6</td>
</tr>
</tbody>
</table>
Sounds Category

The global constants for the multimedia sounds are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snd_Asterisk</td>
<td>#40</td>
</tr>
<tr>
<td>Snd_Beep</td>
<td>-1</td>
</tr>
<tr>
<td>Snd_Default</td>
<td>0</td>
</tr>
<tr>
<td>Snd_Exclamation</td>
<td>#30</td>
</tr>
<tr>
<td>Snd_Hand</td>
<td>#10</td>
</tr>
<tr>
<td>Snd_Question</td>
<td>#20</td>
</tr>
</tbody>
</table>

The waveform sound for each sound type is identified by an entry in the Sounds section of the registry. (Assign sounds to system events by using the Sounds and Multimedia program item of the standard Windows Control Panel.)

SystemEvents Category

The global constants for JADE events for which system notifications are sent are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any_System_Event</td>
<td>0</td>
<td>Object has been created, deleted, or updated</td>
</tr>
<tr>
<td>Object_Create_Event</td>
<td>4</td>
<td>Object has been created</td>
</tr>
<tr>
<td>Object_Delete_Event</td>
<td>6</td>
<td>Object has been deleted</td>
</tr>
<tr>
<td>Object_Update_Event</td>
<td>3</td>
<td>Object has been updated</td>
</tr>
<tr>
<td>System_Base_Event</td>
<td>#80000000</td>
<td></td>
</tr>
</tbody>
</table>

SystemLimits Category

The global constants for JADE system limits are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max_Binary_Byte</td>
<td>#FF</td>
</tr>
<tr>
<td>Max_Identifier_Byte</td>
<td>100</td>
</tr>
<tr>
<td>Max_Integer</td>
<td>#7FFFFFFF (equates to 2,147,483,647)</td>
</tr>
<tr>
<td>Max_Integer64</td>
<td>#7FFFFFFFFFFFFF (equates to 9,223,372,036,854,775,807)</td>
</tr>
<tr>
<td>Max_UnboundedLength</td>
<td>#FFFFFFFF</td>
</tr>
<tr>
<td>MaximumCollectionBlockSize</td>
<td>#40000</td>
</tr>
<tr>
<td>MaximumCollectionDisplaySize</td>
<td>32000</td>
</tr>
<tr>
<td>Min_Binary_Byte</td>
<td>#00 (equates to 0)</td>
</tr>
</tbody>
</table>
Global Constants Reference

**TimerDurations Category**

The global constants for timer durations are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer_Coontinuous</td>
<td>0</td>
<td>Occurs continuously until it is disabled by the Object::endTimer method</td>
</tr>
<tr>
<td>Timer_OneShot</td>
<td>1</td>
<td>Occurs once only</td>
</tr>
</tbody>
</table>

For details, see the Object class beginTimer method, in Chapter 1 of the JADE Encyclopaedia of Classes.

**UUIDVariants Category**

The global constants for specifying the layout of a generated Universally Unique Identifier (UUID) are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VariantDce</td>
<td>2</td>
<td>Distributed Computing Environment, which is the scheme used by Qt C++ application development framework, and which is the recommended variant to pass to the generateUuid method</td>
</tr>
<tr>
<td>VariantMicrosoft</td>
<td>3</td>
<td>Reserved for Microsoft backward compatibility (GUID)</td>
</tr>
<tr>
<td>VariantNcs</td>
<td>1</td>
<td>Reserved for NCS (Network Computing System) backward compatibility</td>
</tr>
</tbody>
</table>

**UserEvents Category**

The global constants for user events are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any_User_Event</td>
<td>-1</td>
<td>-1 (to subscribe to all user events)</td>
</tr>
<tr>
<td>User_Base_Event</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>User_Max_Event</td>
<td>Max_Integer</td>
<td>Max_Integer (#7FFFFFFF, equating to 2147483647)</td>
</tr>
</tbody>
</table>