

# WebTransactions V7.5

Template Language

## **Comments... Suggestions... Corrections...**

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to:

[manuals@ts.fujitsu.com](mailto:manuals@ts.fujitsu.com)

## **Certified documentation according to DIN EN ISO 9001:2008**

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2008.

cognitas. Gesellschaft für Technik-Dokumentation mbH

[www.cognitas.de](http://www.cognitas.de)

## **Copyright and Trademarks**

Copyright © Fujitsu Technology Solutions GmbH 2010.

All rights reserved.

Delivery subject to availability; right of technical modifications reserved.

All hardware and software names used are trademarks of their respective manufacturers.

---

# Contents

<b>1</b>	<b>Preface . . . . .</b>	<b>15</b>
<b>1.1</b>	<b>Product characteristics . . . . .</b>	<b>15</b>
<b>1.2</b>	<b>WebTransactions documentation . . . . .</b>	<b>17</b>
<b>1.3</b>	<b>Structure and target group of this manual . . . . .</b>	<b>19</b>
<b>1.4</b>	<b>New features . . . . .</b>	<b>20</b>
<b>1.5</b>	<b>Notational conventions . . . . .</b>	<b>21</b>
<b>2</b>	<b>Overview of the WTML template language . . . . .</b>	<b>23</b>
<b>2.1</b>	<b>WebLab - WebTransactions development environment . . . . .</b>	<b>23</b>
<b>2.2</b>	<b>Overview of language resources . . . . .</b>	<b>24</b>
<b>2.3</b>	<b>Example: template structure . . . . .</b>	<b>27</b>
<b>3</b>	<b>Lexical elements . . . . .</b>	<b>29</b>
<b>3.1</b>	<b>Character set . . . . .</b>	<b>29</b>
<b>3.2</b>	<b>White space characters . . . . .</b>	<b>30</b>
<b>3.3</b>	<b>Separators . . . . .</b>	<b>31</b>
<b>3.4</b>	<b>Line-end characters . . . . .</b>	<b>31</b>
<b>3.5</b>	<b>Comments . . . . .</b>	<b>31</b>
<b>3.6</b>	<b>Keywords . . . . .</b>	<b>32</b>
<b>3.7</b>	<b>Literals . . . . .</b>	<b>33</b>
3.7.1	Text literals . . . . .	33
3.7.2	Natural numbers . . . . .	34
3.7.3	Floating-point values . . . . .	34

# Contents

---

3.7.4	Strings (string literals) . . . . .	35
3.7.5	Logical values . . . . .	36
3.7.6	Literal for an array object . . . . .	36
3.7.7	Literal for an object . . . . .	36
3.7.8	Literal for the null object . . . . .	37
3.7.9	Literals for regular expressions . . . . .	38
<b>3.8</b>	<b>Name elements . . . . .</b>	<b>42</b>
<b>4</b>	<b>Data types, variables, and names . . . . .</b>	<b>43</b>
<b>4.1</b>	<b>Data types . . . . .</b>	<b>44</b>
4.1.1	number . . . . .	45
4.1.2	boolean . . . . .	45
4.1.3	undefined . . . . .	45
4.1.4	string . . . . .	46
4.1.5	object . . . . .	46
4.1.6	function . . . . .	46
4.1.7	Stringlike data types . . . . .	47
4.1.8	Type conversion . . . . .	47
<b>4.2</b>	<b>Local and global variables . . . . .</b>	<b>49</b>
<b>4.3</b>	<b>Lifetime of variables . . . . .</b>	<b>51</b>
<b>4.4</b>	<b>Initialization . . . . .</b>	<b>53</b>
<b>4.5</b>	<b>Name structure . . . . .</b>	<b>53</b>
4.5.1	Fully qualified specifications . . . . .	54
4.5.2	Relative specifications . . . . .	55
4.5.3	Assigning names to objects . . . . .	56
<b>4.6</b>	<b>User-defined classes . . . . .</b>	<b>57</b>
<b>4.7</b>	<b>Object hierarchy and inheritance . . . . .</b>	<b>59</b>
<b>5</b>	<b>Expressions and operators . . . . .</b>	<b>63</b>
<b>5.1</b>	<b>Different types of expressions . . . . .</b>	<b>64</b>
<b>5.2</b>	<b>Arithmetic operators . . . . .</b>	<b>65</b>
<b>5.3</b>	<b>Comparison operators . . . . .</b>	<b>66</b>

---

<b>5.4</b>	<b>Bitwise operators</b>	<b>68</b>
5.4.1	Bitwise logical operators (&,  , ^, ~)	68
5.4.2	Bitwise shift operators (<<, >>, >>>)	69
<b>5.5</b>	<b>Boolean operators (&amp;&amp;,   , !)</b>	<b>70</b>
<b>5.6</b>	<b>Assignment operators</b>	<b>71</b>
<b>5.7</b>	<b>String concatenation operator (+)</b>	<b>72</b>
<b>5.8</b>	<b>Special operators</b>	<b>73</b>
5.8.1	Condition operator (?:)	73
5.8.2	Comma operator ( , )	74
5.8.3	new operator	75
5.8.4	delete operator	76
5.8.5	in operator	76
5.8.6	instanceof operator	77
5.8.7	WT_THIS (for class templates only)	77
5.8.8	this	77
5.8.9	Evaluation operator ##...#	79
5.8.10	typeof operator	80
5.8.11	void operator	81
<b>5.9</b>	<b>Evaluation sequence</b>	<b>82</b>
<b>6</b>	<b>Global functions</b>	<b>83</b>
<hr/>		
<b>6.1</b>	<b>copyFile() function</b>	<b>83</b>
<b>6.2</b>	<b>createFolder() function</b>	<b>84</b>
<b>6.3</b>	<b>deleteFile() function</b>	<b>85</b>
<b>6.4</b>	<b>escape() function</b>	<b>86</b>
<b>6.5</b>	<b>eval() function</b>	<b>87</b>
<b>6.6</b>	<b>evaluate() function</b>	<b>88</b>
<b>6.7</b>	<b>exitDialogStep() function</b>	<b>90</b>
<b>6.8</b>	<b>exitReceiveProcessing() function</b>	<b>91</b>
<b>6.9</b>	<b>exitScript() function</b>	<b>92</b>
<b>6.10</b>	<b>exitSession() function</b>	<b>94</b>
<b>6.11</b>	<b>exitTemplate() function</b>	<b>95</b>
<b>6.12</b>	<b>forward() function</b>	<b>96</b>

## Contents

---

6.13	<b>import function()</b> . . . . .	98
6.14	<b>include() function</b> . . . . .	99
6.15	<b>isRequestWaiting() function</b> . . . . .	102
6.16	<b>listFolder() function</b> . . . . .	104
6.17	<b>moveFile() function</b> . . . . .	106
6.18	<b>Number() function</b> . . . . .	107
6.19	<b>parseFloat() function</b> . . . . .	108
6.20	<b>parseInt() function</b> . . . . .	109
6.21	<b>setNextPage() function</b> . . . . .	110
6.22	<b>setSingleStep() function</b> . . . . .	111
6.23	<b>setTimeout() function</b> . . . . .	112
6.24	<b>setTraceLevel() function</b> . . . . .	113
6.25	<b>String() function</b> . . . . .	113
6.26	<b>unescape() function</b> . . . . .	114
6.27	<b>writeToTrace() function</b> . . . . .	115
<b>7</b>	<b>Built-in classes and methods</b> . . . . .	<b>117</b>
<b>7.1</b>	<b>Array class</b> . . . . .	<b>118</b>
7.1.1	Constructors . . . . .	118
7.1.2	Attributes . . . . .	120
7.1.3	concat method . . . . .	121
7.1.4	equals method . . . . .	122
7.1.5	getClassName method . . . . .	123
7.1.6	join method . . . . .	124
7.1.7	pop method . . . . .	125
7.1.8	push method . . . . .	126
7.1.9	reverse method . . . . .	127
7.1.10	shift method . . . . .	128
7.1.11	slice method . . . . .	129
7.1.12	sort method . . . . .	130
7.1.13	splice method . . . . .	133
7.1.14	toString method . . . . .	134
7.1.15	unshift method . . . . .	135
7.1.16	valueOf method . . . . .	136

---

<b>7.2</b>	<b>Boolean class</b>	<b>137</b>
7.2.1	Constructors	137
7.2.2	equals method	137
7.2.3	getClassName method	138
7.2.4	setValue method	138
7.2.5	toString method	139
7.2.6	valueOf method	140
<b>7.3</b>	<b>Date class</b>	<b>141</b>
7.3.1	Constructors	141
7.3.2	equals method	142
7.3.3	getClassName method	142
7.3.4	get... methods	143
7.3.5	getTimezoneOffset method	144
7.3.6	set... methods	145
7.3.7	toGMTString method	145
7.3.8	toLocaleString method	146
7.3.9	toString method	146
7.3.10	valueOf method	147
<b>7.4</b>	<b>Document class</b>	<b>148</b>
7.4.1	Constructor	148
7.4.2	clear method	149
7.4.3	close method	149
7.4.4	equals method	150
7.4.5	getClassName method	150
7.4.6	open method	151
7.4.7	read method	152
7.4.8	valueOf method	152
7.4.9	write / writeln method	153
<b>7.5</b>	<b>Host data object class</b>	<b>154</b>
7.5.1	getClassName method	154
7.5.2	toString method	155
7.5.3	valueOf method	156
<b>7.6</b>	<b>Function class</b>	<b>157</b>
7.6.1	Constructors	157
7.6.2	Attributes	158
7.6.3	equals method	160
7.6.4	getClassName method	160
<b>7.7</b>	<b>Math class</b>	<b>161</b>
7.7.1	Class attributes	161
7.7.2	abs method	161
7.7.3	acos method	162

# Contents

---

7.7.4	asin method . . . . .	162
7.7.5	atan method . . . . .	163
7.7.6	ceil method . . . . .	163
7.7.7	cos method . . . . .	164
7.7.8	exp method . . . . .	164
7.7.9	floor method . . . . .	165
7.7.10	log method . . . . .	165
7.7.11	max method . . . . .	166
7.7.12	min method . . . . .	166
7.7.13	pow method . . . . .	167
7.7.14	random method . . . . .	167
7.7.15	round method . . . . .	168
7.7.16	sin method . . . . .	169
7.7.17	sqrt method . . . . .	169
7.7.18	tan method . . . . .	169
<b>7.8</b>	<b>Number class . . . . .</b>	<b>170</b>
7.8.1	Constructors . . . . .	170
7.8.2	Class attributes . . . . .	170
7.8.3	equals method . . . . .	171
7.8.4	getClassName method . . . . .	171
7.8.5	setValue method . . . . .	172
7.8.6	toString method . . . . .	172
7.8.7	valueOf method . . . . .	172
<b>7.9</b>	<b>Object class . . . . .</b>	<b>173</b>
7.9.1	Constructors . . . . .	173
7.9.2	equals method . . . . .	174
7.9.3	getClassName method . . . . .	174
7.9.4	toString method . . . . .	175
7.9.5	valueOf method . . . . .	179
<b>7.10</b>	<b>RegExp class . . . . .</b>	<b>180</b>
7.10.1	Constructors . . . . .	180
7.10.2	Attributes of objects of the RegExp class . . . . .	182
7.10.3	Predefined RegExp object . . . . .	183
7.10.4	compile method . . . . .	184
7.10.5	equals method . . . . .	185
7.10.6	exec method . . . . .	186
7.10.7	getClassName method . . . . .	189
7.10.8	test method . . . . .	190
<b>7.11</b>	<b>String class . . . . .</b>	<b>191</b>
7.11.1	Constructors . . . . .	191
7.11.2	Attributes . . . . .	191
7.11.3	charAt method . . . . .	192



7.11.4	charCodeAt method . . . . .	193
7.11.5	concat method . . . . .	194
7.11.6	equals method . . . . .	195
7.11.7	fromCharCode method . . . . .	195
7.11.8	getClassName method . . . . .	196
7.11.9	indexOf method . . . . .	197
7.11.10	lastIndexOf method . . . . .	198
7.11.11	match method . . . . .	199
7.11.12	replace method . . . . .	201
7.11.13	search method . . . . .	203
7.11.14	setValue method . . . . .	204
7.11.15	slice method . . . . .	205
7.11.16	split method . . . . .	206
7.11.17	substr method . . . . .	207
7.11.18	substring method . . . . .	208
7.11.19	toLowerCase method . . . . .	209
7.11.20	toString method . . . . .	209
7.11.21	toUpperCase method . . . . .	210
7.11.22	valueOf method . . . . .	210
<b>7.12</b>	<b>WT_Communication class . . . . .</b>	<b>211</b>
7.12.1	Constructors . . . . .	211
7.12.2	close method . . . . .	212
7.12.3	equals method . . . . .	212
7.12.4	getClassName method . . . . .	213
7.12.5	getModule method . . . . .	213
7.12.6	open method . . . . .	214
7.12.7	receive method . . . . .	214
7.12.8	send method . . . . .	215
<b>7.13</b>	<b>WT_Filter class . . . . .</b>	<b>216</b>
7.13.1	dataObjectToXML method . . . . .	217
7.13.2	dataObjectToFormattedXML method . . . . .	219
7.13.3	methodCallToXML method . . . . .	222
7.13.4	objectTreeToXML method . . . . .	223
7.13.5	XMLToDataObject method . . . . .	224
7.13.6	XMLToMethodCall method . . . . .	225
7.13.7	XMLToObjectTree method . . . . .	226
7.13.8	Methode XML_SAXParse . . . . .	227
<b>7.14</b>	<b>WT_LdapConnection class . . . . .</b>	<b>234</b>
7.14.1	Overview of the LDAP directory service . . . . .	234
7.14.2	LDAP error messages . . . . .	235
7.14.3	Constructor . . . . .	236
7.14.4	add method . . . . .	237

## Contents

---

7.14.5	bind method . . . . .	238
7.14.6	bindSasl method . . . . .	239
7.14.7	compare method . . . . .	240
7.14.8	deleteEntry method . . . . .	241
7.14.9	equals method . . . . .	241
7.14.10	explodeDn method . . . . .	242
7.14.11	firstEntry method . . . . .	243
7.14.12	getClassName method . . . . .	244
7.14.13	getDn method . . . . .	244
7.14.14	getEntries method . . . . .	245
7.14.15	getOption method . . . . .	246
7.14.16	modify method . . . . .	247
7.14.17	nextEntry method . . . . .	248
7.14.18	search method . . . . .	249
7.14.19	setOption method . . . . .	253
7.14.20	toString method . . . . .	254
7.14.21	unbind method . . . . .	254
7.14.22	valueOf method . . . . .	255
7.14.23	WebTransactions and LDAP: examples . . . . .	255
<b>7.15</b>	<b>WT_Userexit class . . . . .</b>	<b>258</b>
7.15.1	Constructors . . . . .	258
7.15.2	Methods . . . . .	259
<b>8</b>	<b>WTML tags . . . . .</b>	<b>261</b>
<hr/>		
<b>8.1</b>	<b>Rem - inserting comments . . . . .</b>	<b>264</b>
<b>8.2</b>	<b>Dataform - defining form areas . . . . .</b>	<b>265</b>
<b>8.3</b>	<b>Exit - terminating processing . . . . .</b>	<b>267</b>
<b>8.4</b>	<b>Include - including templates . . . . .</b>	<b>268</b>
<b>8.5</b>	<b>IF/ELSE/ENDIF control structure . . . . .</b>	<b>269</b>
<b>8.6</b>	<b>DO WHILE loop . . . . .</b>	<b>271</b>
<b>8.7</b>	<b>DO UNTIL loop . . . . .</b>	<b>272</b>
<b>8.8</b>	<b>OnCreateScript - WTScrip at generation time . . . . .</b>	<b>274</b>
<b>8.9</b>	<b>OnReceiveScript - WTScrip after the receipt of browser data . . . . .</b>	<b>275</b>

---

<b>9</b>	<b>WTScript statements (in OnCreateScript/OnReceiveScript)</b> . . . . .	<b>277</b>
<b>9.1</b>	<b>Empty statements</b> . . . . .	<b>278</b>
<b>9.2</b>	<b>Expression as a statement</b> . . . . .	<b>278</b>
<b>9.3</b>	<b>Statement block as a statement</b> . . . . .	<b>279</b>
<b>9.4</b>	<b>Sequence control statements</b> . . . . .	<b>279</b>
9.4.1	if branch . . . . .	279
9.4.2	while loop . . . . .	281
9.4.3	do/while loop . . . . .	283
9.4.4	for loop . . . . .	284
9.4.5	for/in loop . . . . .	286
9.4.6	switch statement . . . . .	288
9.4.7	break statement . . . . .	290
9.4.8	continue statement . . . . .	292
<b>9.5</b>	<b>return statement</b> . . . . .	<b>294</b>
<b>9.6</b>	<b>var statement</b> . . . . .	<b>295</b>
<b>9.7</b>	<b>function statement</b> . . . . .	<b>297</b>
<b>9.8</b>	<b>Function literal</b> . . . . .	<b>300</b>
<b>9.9</b>	<b>with statement</b> . . . . .	<b>301</b>
<b>9.10</b>	<b>Exception handling</b> . . . . .	<b>302</b>
9.10.1	Error object . . . . .	302
9.10.2	Explicit exceptions . . . . .	304
9.10.3	Exception handling procedure . . . . .	305
<b>10</b>	<b>Class templates (*.clt)</b> . . . . .	<b>309</b>
<b>10.1</b>	<b>WT_THIS - accessing the calling object</b> . . . . .	<b>310</b>
<b>10.2</b>	<b>Example: class templates and WT_THIS</b> . . . . .	<b>311</b>
<b>11</b>	<b>Master templates (.wmt)</b> . . . . .	<b>313</b>
<b>11.1</b>	<b>Lines tag</b> . . . . .	<b>315</b>
<b>11.2</b>	<b>Options tag</b> . . . . .	<b>321</b>
11.2.1	Options tag (standard syntax) . . . . .	321
11.2.2	Options tag (extended syntax) . . . . .	323

---

11.3	<b>Rem tag</b> . . . . .	327
11.4	<b>OnReceiveCopies tag</b> . . . . .	328
11.5	<b>GenerationInfo tag</b> . . . . .	329
11.6	<b>Format tag</b> . . . . .	330
11.7	<b>CommObj tag</b> . . . . .	330
11.8	<b>NationalVariant tag</b> . . . . .	330
11.9	<b>GlobalSettings tag</b> . . . . .	331
11.10	<b>Source tag</b> . . . . .	331
11.11	<b>ObjectName tag</b> . . . . .	331
11.12	<b>PackageName tag</b> . . . . .	332
11.13	<b>BinaryFile tag</b> . . . . .	332
11.14	<b>ArchiveName tag</b> . . . . .	332
11.15	<b>MethodInterface tag</b> . . . . .	333
<b>12</b>	<b>Server-side interfaces - Java integration and user exits</b> . . . . .	<b>335</b>
<b>12.1</b>	<b>Java integration in WebTransactions</b> . . . . .	<b>336</b>
12.1.1	Installing the Java runtime environment . . . . .	338
12.1.2	Activating Java support . . . . .	338
12.1.3	Defining parameters for the Java Virtual Machine (JVM) . . . . .	340
12.1.4	Creating Java objects in WTScrip . . . . .	341
12.1.5	Using Java objects in WTScrip . . . . .	342
12.1.6	Accessing class elements . . . . .	343
12.1.7	Invoking Java methods in WTScrip . . . . .	344
12.1.8	Reading and modifying attributes . . . . .	346
12.1.9	Creating and using Java arrays in WTScrip . . . . .	347
12.1.10	Using WTScrip operators with Java objects . . . . .	348
12.1.11	Example . . . . .	349
<b>12.2</b>	<b>Using C/C++ user exits</b> . . . . .	<b>350</b>
12.2.1	Files supplied for supporting C/C++ user exits . . . . .	350
12.2.2	Defining C/C++ user exits . . . . .	351
12.2.3	Linking C/C++ user exits . . . . .	351
12.2.4	Examples of C/C++ user exits . . . . .	353

<b>12.3</b>	<b>Ready-made C/C++ user exits supplied with WebTransactions</b>	<b>355</b>
12.3.1	CheckLogin	357
12.3.2	CheckProcess	357
12.3.3	Creationtime	358
12.3.4	Delfile	359
12.3.5	FreeBuffer	359
12.3.6	FreeNameInPool	359
12.3.7	Getdate	360
12.3.8	Getdir	360
12.3.9	Getfile	360
12.3.10	GetInstallDir	361
12.3.11	Gettime	361
12.3.12	LockNameInPool	362
12.3.13	Modificationtime	362
12.3.14	Putfile	363
12.3.15	ReleaseStationName	363
12.3.16	ReplaceByConfigFile	364
12.3.17	ReserveStationName	364
12.3.18	SendMail	365
12.3.19	WTSleep	366
<b>13</b>	<b>XML conversion</b>	<b>367</b>
<b>13.1</b>	<b>Importing and exporting XML documents</b>	<b>367</b>
13.1.1	Structure of an imported XML object	368
13.1.2	Representation of XML elements	369
<b>13.2</b>	<b>Exporting data structures</b>	<b>372</b>
<b>14</b>	<b>Examples</b>	<b>375</b>
<b>14.1</b>	<b>Changing styles</b>	<b>375</b>
<b>14.2</b>	<b>Polling the Exit button</b>	<b>376</b>
<b>14.3</b>	<b>Saving data with XML conversion</b>	<b>377</b>
<b>15</b>	<b>Short reference guide</b>	<b>379</b>
<b>15.1</b>	<b>WTML tags</b>	<b>379</b>
<b>15.2</b>	<b>WTSript statements (alphabetic order)</b>	<b>380</b>

**Glossary . . . . . 381**

**Abbreviations . . . . . 399**

**Related publications . . . . . 401**

**Index . . . . . 403**

---

# 1 Preface

Over the past years, more and more IT users have found themselves working in heterogeneous system and application environments, with mainframes standing next to Unix systems and Windows systems and PCs operating alongside terminals. Different hardware, operating systems, networks, databases and applications are operated in parallel. Highly complex, powerful applications are found on mainframe systems, as well as on Unix servers and Windows servers. Most of these have been developed with considerable investment and generally represent central business processes which cannot be replaced by new software without a certain amount of thought.

The ability to integrate existing heterogeneous applications in a uniform, transparent IT concept is a key requirement for modern information technology. Flexibility, investment protection, and openness to new technologies are thus of crucial importance.

## 1.1 Product characteristics

With WebTransactions, Fujitsu Technology Solutions offers a best-of-breed web integration server which will make a wide range of business applications ready for use with browsers and portals in the shortest possible time. WebTransactions enables rapid, cost-effective access via standard PCs and mobile devices such as tablet PCs, PDAs (Personal Digital Assistant) and mobile phones.

WebTransactions covers all the factors typically involved in web integration projects. These factors range from the automatic preparation of legacy interfaces, the graphic preparation and matching of workflows and right through to the comprehensive frontend integration of multiple applications. WebTransactions provides a highly scalable runtime environment and an easy-to-use graphic development environment.

On the first integration level, you can use WebTransactions to integrate and link the following applications and content directly to the Web so that they can be easily accessed by users in the internet and intranet:

- Dialog applications in BS2000/OSD
- MVS or z/OS applications
- System-wide transaction applications based on openUTM
- Dynamic web content

Users access the host application in the internet or intranet using a web browser of their choice.

Thanks to the use of state-of-the-art technology, WebTransactions provides a second integration level which allows you to replace or extend the typically alphanumeric user interfaces of the existing host application with an attractive graphical user interface and also permits functional extensions to the host application without the need for any intervention on the host (dialog reengineering).

On a third integration level, you can use the uniform browser interface to link different host applications together. For instance, you can link any number of previously heterogeneous host applications (e.g. MVS or OSD applications) with each other or combine them with dynamic Web contents. The source that originally provided the data is now invisible to the user.

In addition, you can extend the performance range and functionality of the WebTransactions application through dedicated clients. For this purpose, WebTransactions offers an open protocol and special interfaces (APIs).

Host applications and dynamic Web content can be accessed not only via WebTransactions but also by “conventional” terminals or clients. This allows for the step-by-step connection of a host application to the Web, while taking account of the wishes and requirements of different user groups.



## 1.2 WebTransactions documentation

The WebTransactions documentation consists of the following documents:

- An introductory manual which applies to all supply units:

### **Concepts and Functions**

This manual describes the key concepts behind WebTransactions:

- The various possible uses of WebTransactions.
  - The concept behind WebTransactions and the meanings of the objects in WebTransactions, their main characteristics and methods, their interaction and life cycle.
  - The dynamic runtime of a WebTransactions application.
  - The administration of WebTransactions.
  - The WebLab development environment.
- A User Guide for each type of host adapter with special information about the type of the partner application:

### **Connection to openUTM applications via UPIC**

### **Connection to OSD applications**

### **Connection to MVS applications**

All the host adapter guides contain a comprehensive example session. The manuals describe:

- The installation of WebTransactions with each type of host adapter.
- The setup and starting of a WebTransactions application.
- The conversion templates for the dynamic conversion of formats on the web browser interface.
- The editing of templates.
- The control of communications between WebTransactions and the host applications via various system object attributes.
- The handling of asynchronous messages and the print functions of WebTransactions.

- A User Guide that applies to all the supply units and describes the possibilities of the HTTP host adapter:

### **Access to Dynamic Web Contents**

This manual describes:

- How you can use WebTransactions to access a HTTP server and use its resources.
- The integration of SOAP (Simple Object Access Protocol) protocols in WebTransactions and the connection of web services via SOAP.

- A User Guide valid for all the supply units which describes the open protocol, and the interfaces for the client development for WebTransactions:

### **Client APIs for WebTransactions**

This manual describes:

- The concept of the client-server interface in WebTransactions.
- The `WT_RPC` class and the `WT_REMOTE` interface. An object of the `WT_RPC` class represents a connection to a remote WebTransactions application which is run on the server side via the `WT_REMOTE` interface.
- The Java package `com.siemens.webta` for communication with WebTransactions supplied with the product.

- A User Guide valid for all the supply units which describes the web frontend of WebTransactions that provides access to the general web services:

### **Web-Frontend for Web Services**

This manual describes:

- The concept of the web frontend for object-oriented backend systems.
- The generation of templates for the connection of general web services to WebTransactions.
- The testing and further development of the web frontend for general web services.

## 1.3 Structure and target group of this manual

This manual is intended for anyone who wishes to design his/her own WebTransactions applications. It describes all the WTML template language resources available, allowing you to adapt the results of automatic conversion to meet your own needs when integrating host applications in the web.

The manual is designed as a reference guide in which you can look up information quickly. However, it also contains numerous examples which illustrate the presented language resources.

You will find this manual easier to follow if you are already familiar with the basic principles of the HTML markup language. Although knowledge of a high-level programming language is also desirable, it is by no means essential.

The chapters in this manual can be subdivided into four blocks:

- Introduction  
Chapter 2 starts by presenting a brief overview of the concepts on which the template language is based.
- Reference  
Chapters 3 through 12 contain a complete description of all the language resources. Chapter 13 describes the basic concepts of XML conversion.
- Examples  
Chapter 14 contains examples which illustrate the way the language resources interact.
- Overview and reference aids  
The short reference list in chapter 15 is designed to give you an overview and help you track down the required information quickly and easily.


## 1.4 New features

This section describes only those new features which relate to the template language. For a general overview of new features and functions, refer to the WebTransactions manual “Concepts and Functions”.

Type of new feature	Description
Global functions: – New function <code>copyFile()</code> – New function <code>isRequestWaiting()</code> – New function <code>moveFile()</code> – New parameter <code>all</code> in <code>listFolder()</code>	<a href="#">page 83</a> <a href="#">page 102</a> <a href="#">page 106</a> <a href="#">page 104</a>
Built-in classes and methods: – New method <code>string.charCodeAt</code> – New method <code>string.fromCharCode</code> – New method <code>WT_Filter.dataObjectToFormattedXML</code> – Change for the output of the <code>toString()</code> method on objects and arrays: better serialization/deserialization	<a href="#">page 193</a> <a href="#">page 195</a>  <a href="#">page 219</a>
Exceptions: New attributes <code>strLine</code> , <code>strColumn</code> and <code>strText</code> at the exception object	<a href="#">page 303</a>
C/C++ user exits: New argument <code>SendMail</code>	<a href="#">page 365</a>

## 1.5 Notational conventions

The following notational conventions are used in this documentation:

typewriter font	Fixed components which are input or output in precisely this form, such as keywords, URLs, file names
<i>italic font</i>	Variable components which you must replace with real specifications
<b>bold font</b>	Items shown exactly as displayed on your screen or on the graphical user interface; also used for menu items
[ ]	Optional specifications; do not enter the square brackets themselves
{ <i>alternative1</i>   <i>alternative2</i> }	Alternative specifications. You must select one of the expressions inside the braces. The individual expressions are separated from one another by a vertical bar. Do not enter the braces (curly brackets) and vertical bars themselves.
...	Optional repetition or multiple repetition of the preceding components
	Important notes and further information



---

## 2 Overview of the WTML template language

The WTML (WebTransactions Markup Language) template language allows you to apply individual designs to your WebTransactions applications, not only to the visual design of the user interface but also to the processing logic: you can use WTML in order to actively control the dialog with the host application or to integrate multiple host applications within a single web interface.

WebTransactions is therefore more than simply a link module which transfers messages between the browser and the host application. The powerful resources of the template language also permit genuine application integration: WebTransactions thus allows you to redesign your IT infrastructure without having to modify your host applications.

To avoid misunderstandings:

You do not have to do any programming to use WebTransactions. Instead, you can use the WebTransactions conversion tools to connect your host applications to the web without having to program a single line. The conversion tools use default specifications to generate a template from every component of the user interface (page/sheet). For more detailed information on this subject, please refer to the manuals for the individual product variants. You can use these standard templates unchanged or take them as a starting point for your own individual adaptations.

### 2.1 WebLab - WebTransactions development environment

You can use any text editor to edit templates. However, the WebTransactions development environment, WebLab, offers a particularly convenient way of processing templates. With WebLab you can, for example, insert WTML language resources in a template by simply clicking with the mouse, or define default operations via menus and dialog boxes. WebLab also allows you to modify templates “on the fly” while your WebTransactions application is running. The current objects are displayed graphically and can be modified directly.

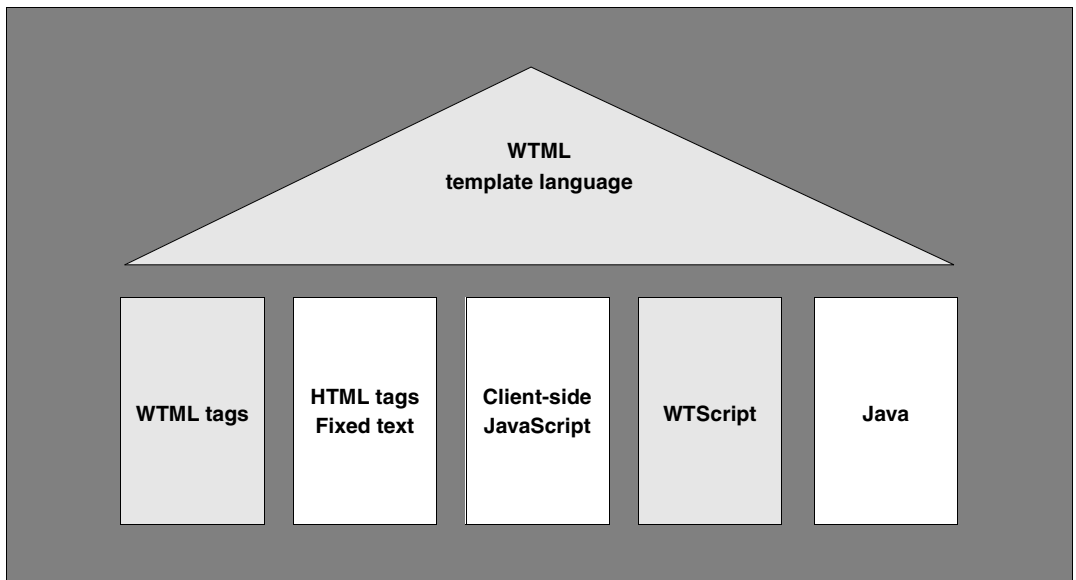


For basic information on WebLab, please refer to the WebTransactions manual “Concepts and Functions”. A detailed description is available in WebLab’s online help function

## 2.2 Overview of language resources

The figure below presents the language resources which you can use when actively designing your templates. The different colors are intended to indicate that HTML and client-side JavaScript are not inflexible components of WTML: the HTML or JavaScript resources which you use here do not depend on WebTransactions but on the employed Web browser. This means that you can always implement the most recent HTML and JavaScript features in your templates. For an example, you can use VBScript or JScript in the Microsoft Internet Explorer. All the tools available for designing web pages are also available for templates.

Although HTML tags and client-side JavaScript are described briefly below, they are not described separately in this manual.



### Standard HTML tags and text

In templates, you can use all the HTML tags and constant texts which the user's browser can interpret. These HTML areas of the template (HTML tag and text) are sent to the browser unchanged.



## JavaScript (client-side)

You can use **all** HTML tags in your templates, i.e. including `<SCRIPT>` tags. This means that you can make use of all the JavaScript language components which can be interpreted by the browser used. Since scripts such as these, like the standard HTML tags, are not interpreted by WebTransactions but by the browser, this is sometimes referred to as client-side JavaScript.

These client-side JavaScript scripts are sent unchanged to the browser in the same way as HTML tags and fixed texts. As far as WebTransactions is concerned, they belong to the HTML area.

## WTML tags

WTML tags allow you to generate or modify HTML pages dynamically, and to control your WebTransactions application. They are used, for example, to calculate values or transfer information from the host application.

WTML tags are WebTransactions-specific functions “dressed up as HTML”. In other words, all the functions which WebTransactions provides for the processing of templates and data or for the control of WebTransactions applications are present in the WTML tags. They have the form: `<wt . . .>`. This means that the syntax of the templates is based on the HTML standard. As a result, you can edit templates either with the WebTransactions editor, WebLab, or with an HTML editor provided that this does not attempt to correct unknown tags automatically. You can also view the template offline directly in the browser; in this case, the WTML tags are simply ignored as “unknown” HTML tags.

Areas containing WTML tags are not sent to the browser. Thus, they do not form part of an HTML area.

The WTML tags are described in detail in [chapter “WTML tags” on page 261](#). For an overview, refer to [section “WTML tags” on page 379](#).

## WTScripTs

WTScripT scripts are similar to client-side JavaScript scripts in areas which start and end with special tags. However, instead of HTML `SCRIPT` tags, you use the WTML tags `WTONCreateScript` and `WTONReceiveScript`. This indicates that these scripts are to be executed by WebTransactions rather than by the browser and also identifies the required time of execution. `OnCreate` scripts are executed before the page is sent to the browser. `OnReceive` scripts are not executed until the response has been received from the browser.

WTScripT scripts allow you to use numerous JavaScript language resources (the statements are listed in [chapter “WTScripT statements \(in OnCreateScript/OnReceiveScript\)” on page 277](#)) together with special WebTransactions classes and functions - e.g. for the exchange of messages with host applications (see [section “Host data object class” on page 154](#) and [section “WT\\_Communication class” on page 211](#)).

Like WTML tag areas, WTScripT areas are not sent to the browser and do not therefore form part of the HTML area.

## Evaluation operators

Evaluation operators allow you to analyze expressions, e.g. the values of objects and attributes. When you do this, the evaluation operators are replaced by the current values, i.e. they are used like variables.

You can use evaluation operators in text, in HTML tags, or in WTML tags. They have the form: `## ... #`. You can use any WTScripT expression in an evaluation operator: thus you can determine the value which is to be output directly in an evaluation operator in an operation of any degree of complexity and - if you wish - also assign this value to a variable which then makes the determined value globally available in the template.

Evaluation operators are described in [section “Evaluation operator ##...#” on page 79](#) and in the [chapter “Class templates \(\\*.clt\)” on page 309](#).

## 2.3 Example: template structure

Here is a simple example which illustrates the structure of a template:

```

<html> (1)
<head>
<title>TRAVEL Main Menu</title>
</head>
<body>
<wtinclude Name="header"> (2)
<form WebTransactions name="myForm"> (3)
<wtoncreatescript>
WT_HOST.OSD_0.receive(); (4)
<</wtoncreatescript>
Date: ##WT_Host.COMM1.DATE.Value# <br> (5)
<input type="button" name="SELBUTTON" value="Book" >
<input type="button" name="SELBUTTON" value="Inquire" >
<input type="button" name="SELBUTTON" value="Cancel" >
<input type="button" name="SELBUTTON" value="Connect" >
...
<wtonreceivescript> (6)
host=WT_HOST[WT_SYSTEM.HANDLE]; (7)
switch (WT_POSTED.SELBUTTON)
{
  case "Book":
    host.SELBUTTON.Value = 1;
    break;
  case "Inquire":
    host.SELBUTTON.Value = 2;
    break;
  case = "Cancel":
    host.SELBUTTON.Value = 3;
    break;
  case "Connect":
    host.SELBUTTON.Value = 4;
}
host.send();
</wtonreceivescript> (8)
... (9)
</form> (10)
</body>
</html>

```

(1) The example starts with an HTML area which is passed to the browser unchanged.

- (2) The first WTML tag is a `wtincl` which includes the contents of the file `header.htm` at this point. You can assign any file name, but it must have the suffix `.htm`. This file can contain any text together with HTML and WTML code. For example, you may want to store the definition of a form header in this type of include file to ensure that the generated HTML pages have a uniform “look and feel” which can be easily modified at a later point.
- (3) Next comes the WTML tag `form WebTransactions`. Internally, this is replaced by an HTML `FORM` tag and a series of hidden fields. This tag marks the start of a form area containing data which is to be sent back by the browser to the WebTransactions application.
- (4) The next element is a `wtoncreatescript` script which WebTransactions executes immediately when generating the HTML output. In this example, the `receive` function is called at a communication object. This means that WebTransactions should read a message from the host application at this point. The data received from the host application is then immediately available in the form of host data objects which correspond to the individual fields of the converted format.
- (5) In the following HTML area, an evaluation operator is used to directly access the `Value` attribute of one of these host data objects. The evaluation operator ensures that the ascertained value is integrated at the relevant position in the HTML area. A series of buttons are also defined in this HTML area.
- (6) The next tag is another WTML tag: `wtonreceivescript` indicates that a WTScrip area follows. It also indicates that the following WTScrip script is not to be executed immediately, but rather “`onReceive`”, i.e. after the HTML page has been generated and sent to the browser and the response posted by the browser has been received.
- (7) Since WebTransactions “remembers” the steps defined in this WTScrip script and does not execute them until the response has been received from the browser, you can define how the response is to be processed at this stage: the values returned for `SELBUTTON` are evaluated and assigned to the corresponding host data object. The final action is to send the host data objects to the host application.
- (8) The concluding WTML tag `/wtonreceivescript` indicates the end of the WTScrip script.
- (9) Here you could, for example, define further HTML tags in order to generate further buttons or text boxes. In the browser display, these would immediately follow the buttons generated in (5). The buttons defined here in (9) could then also be evaluated by the WTScrip script in (7) or in another `onReceive` script.
- (10) The concluding WTML tag `/form` terminates the form area.



For further information on the dynamic sequence of communications and on the tag processing sequence, see the WebTransactions manual “Concepts and Functions”.

---

## 3 Lexical elements

This chapter describes the lexical elements of the template language:

- Character set ([page 29](#))
- White spaces ([page 30](#))
- Separators ([page 31](#))
- Line-end characters ([page 31](#))
- Comments ([page 31](#))
- Keywords ([page 32](#))
- Literals ([page 33](#))
- Name elements ([page 42](#))

It is also necessary to consider the operators: although operators also belong to the basic lexical elements, they are not described here but in [chapter “Expressions and operators” on page 63ff.](#)

### 3.1 Character set

All the characters in 8-bit characters sets (e.g. ISO character sets such as ISO-8859-1 or Windows character sets such as Windows-1250) are permitted in WebTransactions templates

These character sets contain 256 characters. Of these, the first 128 are identical with the classical 7-bit-ASCII-character set. The second 128 characters contain all the special characters and letters used in certain languages, e.g. "ö", "ß" or "á".

## 3.2 White space characters

The term white space characters covers all the characters which have no significance for WebTransactions and are ignored during the sequential processing of the template. The sole function of white spaces is to make the coding of the template easier to follow for human readers. The space, tab, new line, and form feed characters often act as white spaces.

Whether or not WebTransactions ignores a character as a white space depends on the context: for example, spaces used to indent a statement in an `OnCreate` or `OnReceive` script are white spaces and are therefore ignored. In contrast, a white space which separates a keyword from a name element within such a statement is not a white space but instead functions as a separator.

### *Example*

```
function myfunction(param1,param2) {
    var x = 99999;
    vary = 99999;
    ...
}
```

### *Explanation*

In the first allocation, the keyword `var` is used to define the local variable `x`. In the second allocation, a global variable with the name `vary` is defined. The space between `var` and `x` is therefore meaningful, and is consequently not a white space. The same applies to the space between `function` and `myfunction`. All the other characters are white spaces: they are used enhance the visual appearance and are ignored by WebTransactions.

However, the foregoing is only true if the function is defined within an `OnCreate` or `OnReceive` script. If it appears within a client-side script (between the HTML tags `<SCRIPT>` and `</SCRIPT>`), then it is part of an HTML literal: all the characters within an HTML literal are sent unchanged to the browser. In this case, there are therefore no white spaces.

### 3.3 Separators

The following separators are used:

“meaningful” spacer characters

(spaces, tabs, new lines, form feeds)

() for bracketing text expressions.

{ } for forming blocks.

,

for separating parameters in function definitions and function calls.

Please note that in other contexts the comma functions as an operator (see [section “Comma operator \( , \)” on page 74](#)).

;

to terminate statements.

### 3.4 Line-end characters

Lines are considered to be terminated by the characters CR (carriage return), LF (line feed), or the sequence CR LF (where CR LF represents only **one** end-of-line). As a result, templates created on Unix systems or under Windows can be processed in the same way.

Line-end characters inside `OnCreate` or `OnReceive` scripts are white spaces or separators. WebTransactions recognizes the end of a statement by the presence of the separator “;” which is mandatory as the end-of-statement character in server-side scripts (in client-side scripts, the concluding semicolon can be omitted if the statement is located in a line of its own).

### 3.5 Comments

Comments in HTML areas are identified by `wtRem` tags (see [section “Rem - inserting comments” on page 264](#)).

You can also use `wtRem` tags for comments in WTML tags and script areas in the same way as in HTML areas. In addition, it is possible to use multi-line and one-line comments in JavaScript format. Comments in this format cannot be nested:

```
// Comment (one-line comment)
```

```
/* Comment*/ (one or multi-line comment)
```

## 3.6 Keywords

The following keywords can be used within WTML tags, and are not case-sensitive.

action	toAttribute	wtEndIf
case	toObject	wtExit
default	type	wtIf
fromAttribute	upper	wtInclude
fromObject	value	wtOnCreate
function	wtArgument	wtOnCreateScript
lower	wtDataForm	wtOnReceive
name	wtDo	wtOnReceiveScript
onSubmit	wtDoWhile	wtUntil
scope	wtElse	

The following keywords can be used within `OnCreate` and `OnReceive` script areas. These keywords are case-sensitive.

break	function	try
catch	if	typeofvar
continue	in	void
delete	instanceof	while
do	new	with
else	prototype	WT_THIS
finally	return	
for	this	



Keywords cannot be used as name elements.

Although many other expressions, such as `true` and `false`, initially appear to be keywords, they are technically speaking specific values. However, these values are also impermissible as name elements.



## 3.7 Literals

Literals are used to specify constants directly in templates (“literal” specification of a value).

There are literals for HTML areas, natural numbers, floating-point values, strings, logical values, the null object, and regular expressions.

### 3.7.1 Text literals

Each HTML area consists of a text literal which is passed unchanged to the browser. Text literals may contain any number of HTML tags, fixed texts for output, and client-side JavaScript code.

The characters present in HTML literals are passed unchanged to the browser. The only exception is a backslash at line end which cancels the end-of-line: `\line-end` is replaced by nothing. If in the HTML page you want to output a backslash at line-end or characters which normally terminate the HTML area, use either the following HTML escape sequences or octal or hexadecimal input:

Character for display	in HTML	in client-side Javascript (within a string)
\ at line-end	&#92;end-of-line	\134 or \x5C
##	&#35;&#35;	\43\43 or \x23\x23
<	&lt; or &#60;	\74 or \x3C

#### *Example*

`<p> The evaluation operator has the following form: &#35;&#35;...&#35; </p>`

Output in the browser:

The evaluation operator has the following form: ##...#

### 3.7.2 Natural numbers

Literals for natural numbers (integer values) can be entered in decimal, octal, or hexadecimal notation. The prefix 0 indicates octal notation; 0x or 0X indicates hexadecimal notation:

Type of literal	Format	Max. value
Octal literals	0 <i>octalDigit</i> ...	037777777777
Decimal literals	0   { <i>digit1</i> [ <i>digit</i> ... ] }	4294967295
Hexadecimal literals	0 {x   X} <i>hexDigit</i> ...	0xffffffff

*octalDigit* one of the characters 0–7.

*digit1* one of the characters 1–9.

*digit* one of the characters 0–9.

*hexDigit* one of the characters 0–9, A–F, a–f.

### 3.7.3 Floating-point values

Literals for floating-point values can be specified as sequences of digits with positions before or after the decimal point. Exponential notation is also possible.

A literal for a floating-point value must have the following form:

---

```
{digit ... [digit...] [ {e|E} [+|-]digit ... ] } |
{ [digit ... ] digit ... [ {e|E} [+|-]digit ... ] } |
{digit ... {e|E} [+|-]digit ... }
```

---

where *digit* is one of the characters 0–9.

The number range and the accuracy of the conversion of the literals into the internal representation is machine-dependent.

*Examples*

```
3.1415
31415.e-4
.031415E2
31415E-4
```

### 3.7.4 Strings (string literals)

String literals are enclosed by single (') or double (") quotes. The same type of quote must be used to open and close the literal. If the type of quote used to open and close the literal occurs within the string, then it must be preceded by a backslash to deactivate its quote function.

String literals have the following format:

---

```
"[{char1 | \"}] ... " | ' [{char2 | \'}] ... '
```

---

where *char1* is a character other than " and *char2* is a character other than '.

#### Escape sequences in strings

The backslash (\) is used to initiate an escape sequence which is interpreted as an alternative meaning of a specific character. The table below lists the possible escape sequences:

Escape sequence	Conversion
<code>\b</code>	Backspace BS
<code>\t</code>	Horizontal tab HT
<code>\n</code>	Line feed LF
<code>\f</code>	Form feed FF
<code>\r</code>	Carriage return CR
<code>\"</code>	Double quote "
<code>\'</code>	Single quote'
<code>\line-end</code>	Nothing
<code>\\</code>	Backslash \
<code>\octalDigit...</code>	Character corresponding to octal value
<code>\hexDigit...</code>	Character corresponding to hexadecimal value
<code>\char</code>	For any other character: <i>char</i>

#### Example

```
document.write("the words \n\"apple\"");
document.write(' and "pear"');
```

#### Output:

```
the words
"apple" and "pear"
```

### 3.7.5 Logical values

The possible logical (Boolean) literals are `true` and `false`.

### 3.7.6 Literal for an array object

Arrays can be specified as literals with the following format in assignments or function call arguments:

---

```
[element1, element2, ...]
```

---

#### *Examples*

```
x=[1,2,3];           // x is now an array with three elements 1, 2, and 3
c = f([1,2]);        // f is called with an array containing the elements
                    // 1 and 2
```

### 3.7.7 Literal for an object

Objects can be specified as literals with the following format in assignments or function call arguments:

---

```
{attribute1: value1, attribute2: value2, ... }
```

---

#### *Examples*

```
x={y:1, z:23};      // x is now an object with the attributes y and z,
                    // which are set to the values 1 and 23 respectively
c = f({Name: "Inge Neumann", Division: "Development"});
                    // f is called with a single argument, namely the
                    // object with the attributes Name and Division
```

### 3.7.8 Literal for the null object

The literal `null` refers to the `null` object. The `null` object is a special object which references “no object”. Certain methods, e.g. the string method `match`, return the `null` object if the search pattern is not found. Other methods return the `null` object if errors occur, e.g. the communication object methods `send` and `receive`.

You can also use the `null` literal in queries in order to check whether or not a particular object exists.

#### *Example*

```
<wtOnCreateScript>
<!--
  host = WT_HOST[WT_SYSTEM.HANDLE];
  if (host.WT_SYSTEM != null )
    host_system = host.WT_SYSTEM;
  else
    host_system = WT_SYSTEM;
//-->
</wtOnCreateScript>
```

The `if` statement in this script queries whether there is a connection-specific system object. The query `if(host.WT_SYSTEM)` would have the same meaning. However, the use of the `null` literal makes the meaning of the `if` statement easier to identify for human readers.

### 3.7.9 Literals for regular expressions

Literals for regular expressions are enclosed in slashes (/). Any slashes or backslashes which occur within the slashes must be deactivated by a preceding backslash. Literals for regular expressions must not start with an asterisk and must not be empty, as otherwise they are interpreted as the start of a comment.

Between the slashes you can enter any regular expression as described in the table below. The terminating slash may be followed by an *i* and/or *g*, meaning ignore uppercase/lowercase or global search respectively.

*Example*

```
/pattern/ig
```

Regular expressions are structured as in JavaScript or Perl. The table below summarizes the scope and meaning of the metacharacters for these regular expressions.



If you wish to process regular expressions as string literals, you must deactivate the following metacharacters individually using a backslash:

*Example*

```
str = "Amsterdam| Brussels|Chemnitz|Dortmund";
document.write(str.split("\\\\|"));
```

Output in the browser:

[Amsterdam,Brussels,Chemnitz,Dortmund]

Meta-character	Meaning
\	<p>Deactivates the following metacharacter:</p> <p>For example, the character <code>*</code> is a metacharacter which means zero or more repetitions of the preceding character (e.g. <code>/a*/</code> means no <code>a</code>, a single <code>a</code>, or a sequence consisting of multiple <code>a</code>'s). To search for the character <code>*</code> itself, you must precede it with a backslash (e.g. <code>/b*a/</code> corresponds to the string <code>b*a</code>).</p> <p>Please note that many metacharacters already possess the backslash as one of their components, e.g. <code>\n</code> for line feed. If you do not want to search for a line feed but instead for the character sequence <code>\n</code>, you must precede the sequence with a backslash: <code>\\n</code></p>

Meta-character	Meaning
<code>^</code>	Start of the searched through string or start of the line. The regular expression <code>/^a/</code> corresponds to the <code>a</code> in the string "another b", but not to the <code>a</code> in the string "Another a".
<code>\$</code>	End of the searched through string or start of the line  For example, the regular expression <code>/d\$/</code> corresponds to the <code>d</code> in "head", but not to the <code>d</code> in "header".
<code>.</code>	Any character apart from a line feed.  <code>/.T/</code> , for example, only corresponds to the <code>CT</code> in "WEB\nTRANSACTIONS", but not to <code>\nT</code> .
<code>[chars]</code>	One of the characters in <i>chars</i> . You can use a hyphen to specify a character range.  <code>/[abcdefg]/</code> means the same as <code>/[a-g]/</code> and corresponds to the <code>g</code> in "spring".
<code>[^chars]</code>	None of the characters in <i>chars</i> . You can use a hyphen to specify a character range. This type of regular expression corresponds to all the characters which are <b>not</b> specified.  <code>/[^abcdefg]/</code> thus means the same as <code>/[^a-g]/</code> and corresponds to all the characters in "spring" with the exception of the <code>g</code> . <code>/[^1-5]/</code> matches <code>x</code> and <code>6</code> in "x246".
<code> </code>	Separation of multiple alternatives.  <code>/Dampf Diesel/</code> thus matches <code>Dampf</code> in "Dampfschiff" and <code>Diesel</code> in "Diesel-schiff".
<code>( )</code>	For grouping search pattern components and storing the corresponding sections of the target: you can access the components of the target which match the bracketed search pattern components via the components of the result array or via the <code>\$1,...,\$9</code> attributes of the predefined object <code>RegExp</code> .  For example, in the string "Dampfschiffahrtsgesellschaft", <code>/((Dampf)((schiff)(fahrt)))/</code> matches <code>Dampfschiffahrt</code> . The components of the target which correspond to the four bracketed partial expressions are then stored in the attributes of the <code>RegExp</code> object: <code>\$1: Dampf</code> <code>\$2: schiffahrt</code> <code>\$3: schiff</code> <code>\$4: fahrt</code>

Meta-character	Meaning
<code>\n</code>	Is a reference to the <i>n</i> -th bracketed search pattern component, where <i>n</i> is a positive integer.  In <code>/((Dampf)((schiff)(fahrt)).*\3/</code> for example <code>\3</code> stands for the 3rd search pattern component ( <code>schiff</code> ). Thus in the string "Dampfschiffahrt oder Dieselschiffahrt" the search pattern corresponds to "Dampfschiffahrt oder Dieselschiff".
<code>*</code>	Preceding expression any number of times (not at all, once, or multiple times).  <code>/ba*/</code> thus corresponds to <code>ba</code> in "bang!", <code>baaaa</code> in "baaaang!", as well as <code>b</code> in "bong!".
<code>+</code>	Preceding expression at least once (once or multiple times).  <code>/ba+/</code> thus corresponds to <code>ba</code> in "bang!" and <code>baaaa</code> in "baaaang!" but <b>not</b> to the <code>b</code> in "bong!".
<code>?</code>	Preceding expression not at all or once.  <code>/ba?/</code> thus corresponds to <code>ba</code> in "bang!", to <code>ba</code> in "baaaag!" and to <code>b</code> in "bong!".
<code>{n}</code>	Preceding expression <i>n</i> times (where <i>n</i> is a positive whole number).  <code>/ba{3}/</code> thus corresponds to <code>baaa</code> in "baaaaag!" but has no match in either "bang!" or "bong!".
<code>{n, }</code>	Preceding expression at least <i>n</i> times (where <i>n</i> is a positive whole number).  <code>/ba{3, }/</code> thus corresponds to <code>baaaaa</code> in "baaaaag!" but has no match in "baang!".
<code>{n, m}</code>	Preceding expression <i>n</i> to <i>m</i> times (where <i>n</i> and <i>m</i> are positive whole numbers).  <code>/ba{2,3}/</code> thus corresponds to <code>baaa</code> in "baaaaag!" and to <code>baa</code> in "baang", but has no match in "bang!".
<code>\t</code>	Tabulator.
<code>\n</code>	New line.
<code>\r</code>	Carriage return.
<code>\f</code>	Form feed.
<code>\v</code>	Vertical tab.
<code>\octalDigit...</code>	Octal number (where <i>octalDigit</i> is a digit between 0 and 7). This specification allows you to embed octal escape sequences for ASCII characters in regular literals.



Meta-character	Meaning
<code>\xhexDigit...</code>	Hexadecimal number (where <i>hexDigit</i> is one of the characters 0–9, A–F, or a–f). This specification allows you to embed hexadecimal escape sequences for ASCII characters in regular literals.
<code>\cA</code>	Where <i>A</i> specifies a control character to be searched for.  <code>\cJ</code> thus corresponds to a line feed and <code>\cM</code> to a carriage return.
<code>\b</code>	Word boundary.  Thus in the string "WebTransactions", <code>/.ns\b/</code> corresponds to <code>ons</code> but not to <code>ans</code> .
<code>\B</code>	No word boundary.  Thus in the string "WebTransactions", <code>/.ns\B/</code> corresponds to <code>ans</code> but not to <code>ons</code> .
<code>\w</code>	One of the characters <code>a-z A-Z 0-9</code> or underscore( <code>_</code> ). <code>\w</code> thus means the same as <code>[A-Za-z0-9_]</code> .  In <code>!x my_Array 77</code> , <code>/\w+/</code> corresponds to <code>x</code> , <code>my_Array</code> , and <code>77</code> .
<code>\W</code>	None of the characters <code>a-z A-Z 0-9</code> or underscore( <code>_</code> ). <code>\W</code> thus means the same as <code>[^A-Za-z0-9_]</code> .  In <code>!x my_Array 77</code> , <code>/\W+/</code> corresponds to <code>!</code> and to the two spaces (before <code>myArray</code> and before <code>77</code> ).
<code>\s</code>	Single white space character (blank, horizontal/vertical tab, carriage return, form or line feed). <code>\s</code> thus means the same as <code>[\t\v\r\n\f]</code> .  In <code>"tool bar"</code> , <code>/\s\w+/</code> corresponds to <code>" bar"</code> .
<code>\S</code>	Any character apart from a white space character. <code>\S</code> thus means the same as <code>[^\t\v\r\n\f]</code> .  In <code>"tool bar"</code> , <code>/\S\w+/</code> corresponds to <code>"tool"</code> and to <code>"bar"</code> .
<code>\d</code>	A digit (one of the characters 0–9). <code>\d</code> means the same as <code>[0-9]</code> .  In <code>"57a"</code> , <code>/\d/</code> therefore corresponds to <code>5</code> and <code>7</code> .
<code>\D</code>	Not a digit (none of the characters 0–9). <code>\D</code> means the same as <code>[^0-9]</code> .  In <code>"57a"</code> , <code>/\D/</code> therefore corresponds to <code>a</code> .

## 3.8 Name elements

Name elements are basic elements which are used to form names.

In the case of simple names, a name element corresponds to a name (e.g. “x” in `x=9`). In the case of qualified names, the name consists of multiple name elements including either the point operator (e.g. `myarray.length`) or the index operator (e.g. `myarray[3]`, `myarray["length"]`). Since such qualified names are not basic lexical elements but rather combinations of these elements, they are not discussed in this chapter but in [section “Name structure” on page 53](#).

### Structure of name elements

The following characters are permitted in name elements:

A–Z, a–z (uppercase and lowercase)

0–9 (digits)

\_ (underscore)

\$ (dollar character)

The first character of a name element must not be a digit and the name element must not be the same as any keyword. Name elements may be of any length.

### Uppercase/lowercase

Name element names are usually case-sensitive. The name elements for the predefined objects `WT_SYSTEM`, `WT_HOST` and `WT_POSTED` represent an exception and can be specified in uppercase or lowercase.

`ResultArray` and `resultarray` therefore designate different variables whereas `Wt_System`, `wt_system`, and `WT_SYSTEM` refer to one and the same object.

### Examples

Permissible name elements: `ResultArray`, `my_array`, `$input1`, `_hits`

---

## 4 Data types, variables, and names

Variables are named storage areas in which you can store data which you require in your template. The contents stored in a variable are known as its “value”.

Alongside its value, every variable is of a certain type. The template language type concept corresponds to that of JavaScript (see [section “Data types” on page 44](#)).

Just as in JavaScript, the template language offers you far more freedom than many other programming languages in the way you handle variables:

- You do not have to declare variables separately.
- The template language uses “loose typing”: even when you explicitly declare a variable using a `var` statement, you do not specify its type. The variable’s type is determined dynamically by the type of value assigned to it.

Thus you can change both the value and type of a variable at any time:

```
a=10; // typeof a is number
a="10"; // typeof a is string
```

- You do not need to use different variable types in the case of natural numbers (integers) and floating-point values. The `number` type is suitable for both kinds of value.
- Data types are converted automatically if required at runtime (see [section “Type conversion” on page 47](#)). For example, the contents of numeric variables can be integrated into output strings without any explicit conversion (see also examples on [page 72](#)):

```
a=4+2;
document.write("The result is: " + a);
```

## 4.1 Data types

Like every literal or constant value, each variable possesses a certain data type. The template language type concept corresponds to that used in JavaScript.

The following data types exist: `undefined`, `number`, `boolean`, `string`, `object`, and `function`. These are presented in sections [“number” on page 45](#) through [“function” on page 46](#). Section [“Stringlike data types” on page 47](#) explains the term “stringlike”, while [section “Type conversion” on page 47](#) describes the rules which WebTransactions uses to perform automatic type conversion.

### Simple data types and reference data types

The data types `number`, `boolean`, and `undefined` are frequently referred to as simple data types, whereas the data types `string`, `object`, and `function` are all known as reference data types. The reason for this is as follows:

In the case of simple data types, the value of the variable is “simply” stored at the storage location which is symbolically designated by the variable name: for example, if you define five variables with the value 1000, then this value is stored at five different locations.

In the case of a variable with a reference data type, however, the internal value is the address of the value itself or - in other words - a reference: if, for example, you define five different names for a function, the function is nevertheless stored only once in memory. You can then “reference” this function with each of the five names.

### Example: The different data types

```
document.writeln (typeof a); // Output: undefined (not yet allocated)
a=4.118;
document.writeln (typeof a); // Output: number
a="Peter";
document.writeln (typeof a); // Output: string
a=false;
document.writeln (typeof a); // Output: boolean
a=new Array();
document.writeln (typeof a); // Output: object
function myfunction() {      // Function definition
    return 10 }
a=myfunction;
document.writeln (typeof a); // Output: function
document.writeln (a());      // Output: 10 (Calls the function under
                             // the new name a)
```

### 4.1.1 number

The `number` data type covers both natural numbers and floating-point values. As in JavaScript, the arithmetic processing of floating-point values is based on the IEEE 754-1985 standard (IEEE, New York).

The values which can be displayed at the computer lie in the range 4.94065645841247e-324 and 1.79769313486232e+308.

There are also certain special numbers:

`NaN` (not a number)

This value is returned as the result of undefined arithmetic operations, for example if you attempt to multiply two strings such as “Peter” and “Mary”.

`NaN` is not part of the linear numberline: the comparison operators `==`, `<`, `<=`, `>`, and `>=` return `false` if one or both of the operands is `NaN`; `!=` returns `true` if one or both of the operands is `NaN`.

`-Infinity` (minus infinity)

This value is returned when a number is lower than the smallest negative number which can be represented.

`Infinity` (plus infinity)

This value is returned when a number is greater than the largest number which can be represented.

### 4.1.2 boolean

The `boolean` data type can assume either of the two logical values `true` and `false`. You can apply the Boolean operators to operands of type `boolean` (see [section “Boolean operators \(&&, ||, !\)” on page 70](#)).

### 4.1.3 undefined

Any variable to which no value has as yet been assigned has the type `undefined` and the value `undefined`.

#### 4.1.4 string

A string is a sequence of ASCII characters. Every string has a predefined length attribute (`string.length`) which specifies the number of characters. There are a number of predefined methods for objects of the `String` class (see [section “String class” on page 191](#)) which, after conversion, also apply to the type `string`.

You can use the `+` operator to link operands of type `string` (see [section “String concatenation operator \(+\)” on page 72](#)).

#### 4.1.5 object

A variable of type `object` is a container for named attributes. It is therefore an associative array. The value of an object of this type is the reference to such a container or the `null` reference. For example, if you assign an object to a variable, you subsequently have a new name for this object and not two distinct objects.

The attributes can be of any type. You can use any name element or integer for their names; in the latter case, the attribute may also be known as an index (see [section “Name structure” on page 53](#)).

There is also a special object known as the `null` object:

The `null` object references “no object”. Certain methods, for example the string method `match`, return the `null` object if the search pattern is not found. Other methods return the `null` object if an error occurs, e.g. the communication object methods `send` and `receive`.

You use the `new` operator to create objects and the `delete` operator to delete them (see [page 75](#)).

#### 4.1.6 function

A function is defined by means of a `function` statement or as an object of the `Function` class (see [page 297](#)). This definition creates a function object with the name of the defined function.

### 4.1.7 Stringlike data types

In certain operations, for example string operations, the level of similarity of a data type to a string is important. The data types `string` and `function`, as well as all objects of classes whose `valueOf` method returns a string, are said to be “stringlike” (currently such objects belong either to the `String` class or to a user-defined class).

For more information on the term “stringlike”, refer to the section [“Example: Arithmetical addition compared to string concatenation” on page 72.](#)

### 4.1.8 Type conversion

As already mentioned at the start of this chapter, you have a certain amount of freedom in the way you handle data types. If the data type of an expression is inappropriate in a certain application context, WebTransactions converts the expression to the appropriate data type wherever this is possible. For example, if you try to multiply two strings, these strings are converted to the data type `number`. If the strings are numerical then multiplication is performed as normal. If they are non-numerical, the multiplication returns the result `NaN` (Not a Number) thus indicating an illegal numerical operation.

The table below presents the rules used by WebTransactions for type conversion:

Initial data type	Target data type				
	function	object	number	boolean	string
undefined	Error	null	"NaN"	false	"undefined"
function	-	Error	Error	true	Header <sup>1</sup>
object					
(not null)	Error	-	valueOf/ "NaN"	valueOf/ true	toString/ valueOf
(null)	Error	-	0	false	"null"
number					
(zero)	Error	Number	-	false	"0"
(nonzero)	Error	Number	-	true	Numerical string
(NaN)	Error	Number	-	false	"NaN"
(Infinity)	Error	Number	-	true	"+Infinity"
(-Infinity)	Error	Number	-	true	"-Infinity"
boolean					
(false)	Error	Boolean	0	-	"false"
(true)	Error	Boolean	1	-	"true"

Initial data type	Target data type				
	function	object	number	boolean	string
<b>string</b>					
(empty)	Error	String	0	false	-
(non-empty)	Error	String	Numerical value / "NaN"	true	-
<b>User-defined</b>	In the case of user-defined classes, the data type is determined by the base class. You should therefore refer to the appropriate table entry.				

<sup>1</sup> Only the header of the function is converted and not - unlike JavaScript - the entire program text.

### Notes on table

The individual cells in the table specify the result supplied by WebTransactions when attempting to convert the initial data types entered in the left-hand column into the data types specified in the top row. Where a cell contains two possibilities - separated by a slash - WebTransactions first attempts the first possibility and - if this fails - then attempts the second.

This results in the following:

`undefined, function, object, number boolean, string`

Variable or values of the corresponding data type

`Number, Boolean, String`

Variable or values of the object type of the corresponding class

`toString`

Result of the `toString` method

`valueOf`

Result of the `valueOf` method if this returns a result of the target data type



## 4.2 Local and global variables

All variables declared outside a function are global, irrespective of whether the keyword `var` is used. A variable declared within a function is only local if you declare it using the keyword `var`. Otherwise it is automatically global.

A global variable is valid everywhere in the WTSript code.

A local variable is valid within the whole of the function in which it is declared, regardless of where in the function it has been defined.

If you declare a variable in a function using `var` and already have a global variable (or in the case of nested functions, a local variable in the outer function) with the same name, the local variable is used within the function, thus the local variable takes priority over the global one. Outside the function, the global variable is addressed.

### *Example 1*

```
<wtoncreatescript>
<!--
var scope = "global";           // Global variable
function test_scope()
{
var scope = "local";           // Homonymous local variable
document.write("Scope=" + scope + "<br>"); // Local variable is used
}
test_scope();                  // Outputs "Scope=local"
//-->
</wtoncreatescript>
```

If the context in which a function is used is not fully known, it is recommended that you always use the keyword `var`. You can thus ensure that global variables are not overridden. The example below shows what happens when you do not use the `var` keyword.

### *Example 2*

```
<wtoncreatescript>
<!--
scope = "global";              // Global variable
function test_scope()
{
    scope = "local";           // Global variable is
                                // overwritten !!!
    document.write("Scope=" + scope + "<br>");// Global variable is used
    newScope = "local";       // Declares another
                                // global variable
}
-->
```

```
test_scope() ; // Outputs "Scope=local"  
document.write("Scope=" + scope + "<br>"); // Outputs "Scope=local"  
document.write("Scope=" + newScope + "<br>"); // Outputs "Scope=local"  
  
//-->  
</wtoncreatescript>
```

You can nest function calls. Since each function has its own local scope, it is possible to have a number of nested scopes. If the called function is defined within the calling function, the called function has access to the global and local variables (and arguments) of the calling function. If the functions are defined independently of each other, the local variables of the calling function are hidden from the called function.

### *Example 3*

```
<wtoncreatescript>  
<!--  
scope = "global"; // Global variable  
function test_scope()  
{  
    var scope = "local"; // Homonymous local  
                        // variable  
    function nested()  
    {  
        var scope="nested"; // Local in nested  
                            // function  
        document.write("Scope=" + scope + "<br>"); // Outputs "Scope=nested"  
    }  
    nested();  
    document.write("Scope=" + scope + "<br>"); // Outputs "Scope=local"  
}  
test_scope();  
document.write("Scope=" + scope + "<br>"); // Outputs "Scope=global"  
  
//-->  
</wtoncreatescript>
```

## 4.3 Lifetime of variables

In WebTransactions, as in JavaScript, there are four rules for the lifetime of variables:

- **Global variables:**  
Global variables created by means of WTML tag actions or in the template's script areas survive from their creation until the last `OnReceiveScript` script of the template has been processed, or until they are explicitly deleted by means of the `delete` operator.
- **Object attributes:**  
Attributes survive for as long as the object to which they belong or until they are explicitly deleted.
- **Current parameters:**  
The current parameters of a function call survive from the moment the function is called until the function ends or until they are explicitly deleted.
- **Local variables of a function:**  
All variables defined within a function using the keyword `var` are local variables of this function. They survive from the moment they are created until the function ends, or until they are explicitly deleted.

### Lifetime of predefined objects

With WebTransactions, you can not only define your own variables; there are also a number of predefined objects to which special rules apply:

- **System object:**  
WebTransactions creates the system object as a global object of type `object` with the name `WT_SYSTEM`. This system object survives for the entire duration of the session. It possesses a number of attributes which are of importance for the control of WebTransactions.
- **Posted object:**  
WebTransactions creates the posted object as a global object of type `object` with the name `WT_POSTED`. This posted object survives for the entire duration of the session. Its attributes are given by the data most recently sent by the browser.
- **Host root object:**  
The predefined object `WT_HOST` is a container for all the communication objects. It survives for the entire duration of the session.
- **Communication objects:**  
A communication object is created as an attribute of `WT_HOST` by the constructor call `WT_Communication`. This communication object survives for the entire duration of the session.

Communication objects can therefore survive for multiple dialog steps. They allow you to handle parallel connections, and thus to integrate multiple host applications within a single WebTransactions application.

- **Host data objects:**

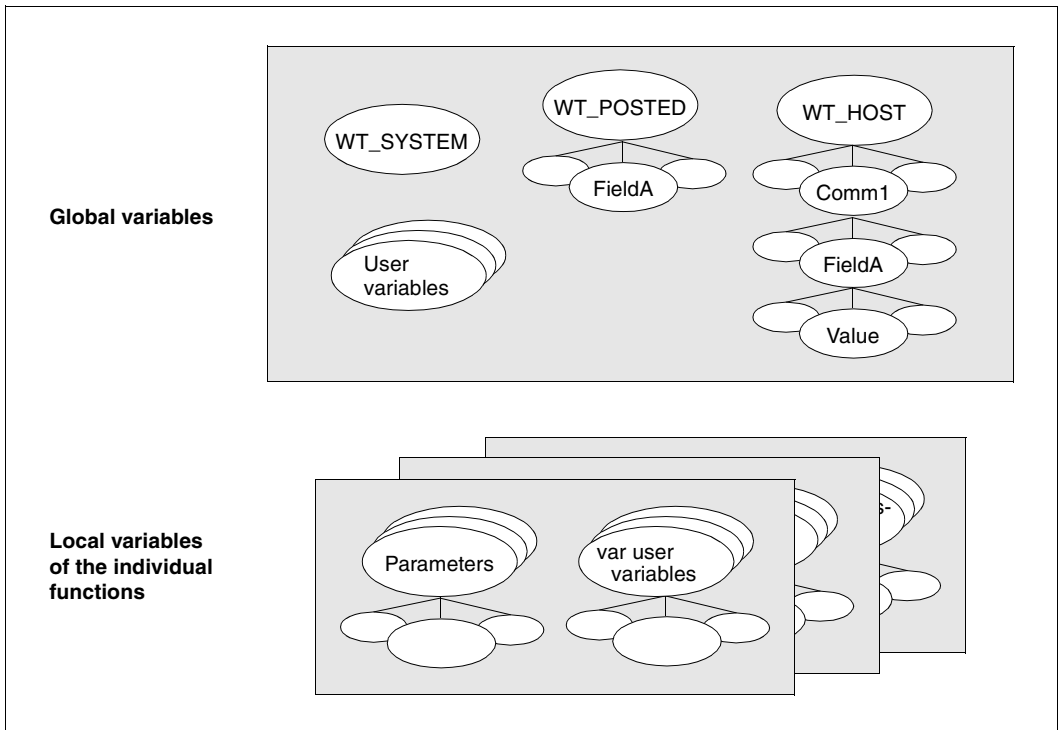
Host data objects represent the sections (e.g. fields) of the actual message body which WebTransactions exchanges with the host. They are created by the method `receive`. When new host data objects are created, older objects are destroyed.



For detailed information on these predefined objects, please refer to the WebTransactions manual “Concepts and Functions”.

**Overview: name spaces for variables**

The figure below illustrates the name spaces. The predefined objects are located together with the global variables created in the script in a global variable space. There is a local variable space for each function currently called. This space contains the local user variables declared using the keyword `var`, together with the current parameters for the function.



## 4.4 Initialization

In WebTransactions, every variable has a value. Certain system object attributes are initialized by WebTransactions at start time. Communication and host data objects are initialized by the communication modules. You can assign a value to initialize any variables which you have created yourself. Any variable to which no value has been assigned has the type `undefined` and the value `undefined`. A parameter in a function call is assigned the current value from the function call. If the call contains no corresponding argument, then the current parameter has the type and value `undefined`.

## 4.5 Name structure

Names designate variables and their substructures as well as functions. There are both simple and compound names. A simple name consists of a single name element. A compound name consists of a sequence of name elements separated by point operators or index operators.

### Point operator

Since name elements must not start with a digit, no index specifications are possible after the point operator.

---

*name element.name element*

---

### Examples

```
myarray.length  
WT_HOST.KOMM1.Command.Value
```

### Index operator

The index operator `[ ]` makes it possible to access all an object's attributes.

---

*name element[expression]*

---

If the square brackets contain an expression which returns a whole number, then a reference to the corresponding index is issued:

*name element[index]*

If the square brackets contain an expression which returns a string, then a reference to the corresponding attribute is issued:

`name element1["name element2"]` and `name element1.name element2` therefore have the same meaning.

### Example

```
for (i=0 ; i < myarray["length"] ; i++) document.writeln(myarray[i]);
```

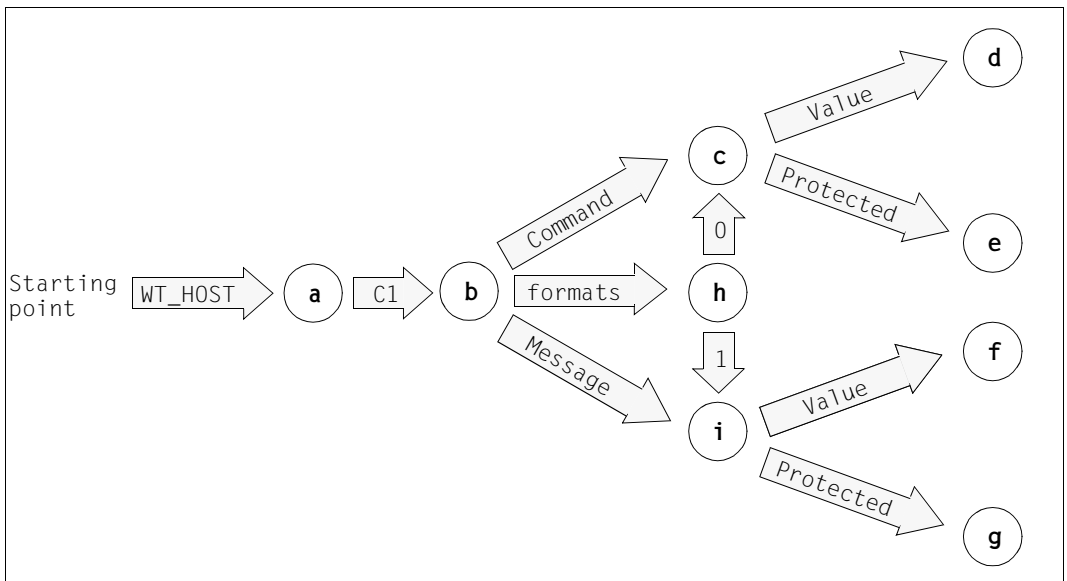
This for loop outputs all the elements of the array `myarray`.

## 4.5.1 Fully qualified specifications

In the case of fully qualified specifications, all the components of the name are specified. A fully qualified name specification provides a unique identification of the designated object independently of the context. Similarly to fully qualified file names in a hierarchical directory structure, each element is identified by a “path specification” which starts at the “root”.

### Example

You could think of a name element as a signpost which points to the designated element. You then obtain the fully qualified name of the element by joining together all the signposts along the path taken to reach the object by means of point or index operators.



The figure above results in, for example, the following fully qualified names:

For a: `WT_HOST`

For b: `WT_HOST.C1`  
`WT_HOST["C1"]`

For c: `WT_HOST.C1.Command`  
`WT_HOST.C1["Command"]`  
`WT_HOST.C1.formats[0]`

For d: `WT_HOST.C1.Command.Value`  
`WT_HOST.C1["Command"]["Value"]`  
`WT_HOST.C1.formats[0].Value`

For g: `WT_HOST.C1.formats[1].Protected`  
`WT_HOST.C1["formats"][1]["Protected"]`

## 4.5.2 Relative specifications

Instead of fully qualified names, you can also specify incomplete names.

In this case, the path is not specified all the way from the root, but from a different starting point. The specification therefore designates the object relative to this starting point.

For example, in the case presented on the previous page, `Command.Value` designates object `d` relative to `b`.

Relative to `c`, `Value` is a name for `d`; relative to `i`, it is a name for `f`.

### 4.5.3 Assigning names to objects

WebTransactions obeys the following rules when assigning an object to the specified name:

1. If the name is located in a `with` statement (see [section “with statement” on page 301](#)), then a check is performed from the inside outwards for each surrounding `with` statement to determine whether the name designates an existing object relative to the object in question. If this is the case, the search is concluded successfully.
2. If the name is located in a function, then a check is performed to determine whether the name designates an object relative to the function object (local variable or parameter of the function). If this is the case, the search is concluded successfully.
3. If steps 1 and 2 fail to yield a positive result, the name is interpreted as a fully qualified name and a check is performed to determine whether a corresponding global object exists. If this is the case, the search is concluded successfully.
4. Finally, a check is performed to determine whether the name designates an object relative to the predefined communication object `WT_HOST.handle` (where *handle* represents the contents of the `HANDLE` attribute of the global system object). If this is the case, the search is concluded successfully.

If, after the application of these rules, no object is found, the name is invalid:

- On a read access, the value `undefined` is returned.
- On a write access then, in the case of single-part names (such as `x` or `result`), an object of this name is implicitly defined.

In the case of multi-part names (such as `result.value`), an initial check is performed to determine whether the name, when reduced to its final component, designates an object of type `object`. If this is the case, then - in the same way as for single-part names - a new object is generated. Otherwise an error is reported.

#### *Example*

```
document.writeln("type of x: " + typeof(x) ); //Output: undefined
x.colour = "red"; //Error because x is not defined as an object
x=new Object("car");
x.colour="red";
document.writeln("now there's a car and its colour is " + x.colour);
```



## 4.6 User-defined classes

Special language components are provided which enable you to create your own classes so that you can define similar objects in WebTransactions. These can be used to define new classes, attributes, and methods.

The procedure for defining your own classes consists of two steps:

1. Describing the class and its attributes
2. Defining the methods

### Describing the class and its attributes

In order to define a new class, you must first define the constructor for objects of this class in the form of a script function. For example:

```
// Constructor for class "Employee":
function Employee() {
  // Definition of class attributes:
  this.name      = "";
  this.division  = "development";
  this.machine   = "computer";
  this.worktime  = 35;
}
```

This defines a constructor for objects of the user-defined class `Employee`. The name of the constructor is also the class name. The keyword `this` is used to define the class attributes and set the default values (in this case, `name`, `division`, `machine`, and `worktime`).

You can now generate objects for this class, in which all attributes are initially set to the default values defined in the constructor. These default values can be overwritten if desired:

```
// An object of the class Employee returns with the attributes
// name = "Manuella Mueller", division = "development",
// machine = "computer" and worktime = 30
angest = new Employee();
angest.name = "Manuella Mueller";
angest.worktime = 30;
```

You can also define additional attributes for individual instances of the new class at any time should this prove necessary for the WebTransactions application. Please note, however, that such attributes apply only to that particular instance.

```
angest.homework = true;
```

## Defining the methods

In order to define a method for a user-defined class, you must first define a script function for this method and then create a reference to this function in the class constructor:

```
// Method for Employee:
function gibName() {
    return (this.name);
}

// Constructor for class "Employee":
function Employee() {
    // Definition of class attributes:
    this.name      = "";
    this.division  = "development";
    this.machine   = "computer";
    this.worktime  = 35;

    // Reference to method:
    this.gibName   = gibName;
}
```

The new method (in this case, `gibName`) can now be used as normal:

```
j = angest.gibName();
```

Such user-defined data types can also be derived from defined classes resulting in an object hierarchy. When objects are derived from classes, they inherit certain properties (attributes and methods). Further information can be found in the following section.

## 4.7 Object hierarchy and inheritance

In `WebTransactions`, it is possible to derive new classes from existing classes or objects. During this process, objects of the new class inherit attributes and methods from the original class. This section contains a step-by-step description of how classes are derived, and points out a number of issues to be noted. We will use the example from the previous section:

```
// Method for Employee:
function gibName() {
    return (this.name);
}

// Constructor for class "Employee":
function Employee() {
    // Definition of class attributes:
    this.name      = "";
    this.division  = "development";
    this.machine   = "computer";
    this.worktime  = 35;

    // Reference to method:
    this.gibName   = gibName;
}
```

This class defines an employee with the same basic properties as all other employees in the company. However, we now need further properties for the different types of employees which vary from one type to the next. For instance, a sales employee may be responsible for different sales regions, or an engineer may be involved in different projects. As the basic properties remain the same for all employees, it makes sense to derive them.

For example, to define an employee of type `SalesManager`, proceed as follows:

```
// Constructor for class "SalesManager"
function SalesManager() {
    // Definition of additional class attributes:
    this.area = 8;    // Sales region, default 8
    this.quota = 100; // Sales quota
}
// Deriving all other attributes from "Employee"
SalesManager.prototype = new Employee();
```

The keyword `prototype` creates a reference to a new object of the class `Employee`, whereby all class attributes from `Employee` are inherited (in the form of a reference) by all new instances of the class `SalesManager`. The keyword `prototype` refers to a real object. In this case, it is not necessary to define a new object of an existing class with `new`. Instead, you can refer to an existing object. The following example illustrates how the class `SalesManager` is derived, but this time from an object:

```
proto = new Object;
proto.name      = "";
proto.division  = "development";
proto.machine   = "computer";
proto.worktime  = 35;

// Deriving the new class "SalesManager" from the object
// "proto"
function SalesManager() {
    // Definition of additional class attributes:
    this.area = 8;      // Sales region, default 8
    this.quota = 100;  // Sales quota
}
// Deriving all other attributes from the object "proto"
SalesManager.prototype = proto;
```

Note that the derived attributes from the prototype object are created initially in the form of references. It is not until you assign a value to such a derived attribute in an instance that an instance attribute is actually created with the new value:

```
manager1 = new SalesManager();

manager1.worktime = 60;           // Instance attribute
document.write("Division   = " + manager1.division + "<BR>");
document.write("Worktime = " + manager1.worktime + "<BR>");
proto.division = "marketing";    // Change in prototype
proto.worktime = 30;             // Change in prototype
document.write("Division   = " + manager1.division + "<BR>");
document.write("Worktime = " + manager1.worktime + "<BR>");
```

This gives the following output:

```
Division = development
Worktime = 60
Division = marketing
Worktime = 60
```



If an attribute of a derived object is deleted explicitly (with the `delete` operator), it is merely identified as deleted in the derived object. It is retained in the prototype object.

It is also possible to derive new classes from predefined classes. This is illustrated in the example below:

```
// Deriving a new class "NamedArray" from the class "Array":
function NamedArray (n) {
    this.name = n;
}
NamedArray.prototype = new Array;

MyArray = new NamedArray("first");
```

The new class `NamedArray` contains all methods of the class `Array` plus the additional attribute `name`. `MyArray` is now an array of length 0 with the name `first`.

As in the `SalesManager` example above, these classes can also be derived directly from an object of the class `Array`:

```
// Deriving a new class "NamedArray" from an "Array" object:
a = new Array;
for (i=0; i<= 10; i++)
    a[i] = 0;

function NamedArray (n) {
    this.name = n;
}
NamedArray.prototype = a;

MyArray = new NamedArray("second");
```

In this case, `MyArray` is an array of length 11 with the name `second` whose elements are preset to 0.



---

## 5 Expressions and operators

This chapter starts by providing an overview of the expressions used in the template language. The following sections [“Arithmetic operators” on page 65](#) through [“Special operators” on page 73](#) present the individual operators. The order in which these operators are evaluated is described at the end of this chapter in [section “Evaluation sequence” on page 82](#).

Expressions are combinations of literals, variables, operators, and expressions, which provide a particular result when evaluated.

The following results are possible in the case of WebTransactions expressions:

- a value  
43+7 for example returns a value of type `number`
- a reference to an object  
For example, an expression which calls a constructor returns a reference to an object:  
`myArray=new Array()`
- undefined  
An expression is `undefined` if an uninitialized variable, a call to a function without a return value, or the `void` operator is used. `undefined` is, however, a value in itself - *all* expressions provide a result.

## 5.1 Different types of expressions

WebTransactions supports all the expressions which are possible in JavaScript. These expressions may occur in the WTScript areas and within an evaluation operator.

- Some expressions assign a value to a variable, whereas others simply possess a value. For example, the expression `x=4+5` assigns the value of the expression `4+5` to the variable `x` and itself represents this value. Such expressions use assignment operators. In contrast, an expression such as `4+5` contains no assignment but simply provides the result 9.
- There are elementary expressions which correspond to lexical units (such as a variable like `x` or a literal such as `42` or “hello world”), and complex expressions which are composed of elementary expressions. In this case, the evaluation rules presented in [section “Evaluation sequence” on page 82](#) apply. However, you may also use brackets `()` to force a particular evaluation sequence.
- Depending on the number of operands linked by an operator, it is common to distinguish between one-position and two-position expressions. Using the condition operator “?:” (see [page 73](#)), it is even possible to form three-position expressions.
- Expressions involving related operators are frequently referred to together using a single term, e.g. a distinction is made between arithmetic expressions and comparison expressions.

### Enhancements compared to JavaScript expressions

The following enhancements have also been incorporated:

- For reasons of compatibility, the conditions of the WTML tags `<wtIf ...>`, `<wtDoWhile ...>`, and `<wtUntil...>` may also contain the comparison operators `#==`, `#!=`, `#>`, `#<`, `#>=`, and `#<=`.
- For reasons of compatibility, strings within WTML tags may contain both fixed characters (string literals) and evaluation operators.



## 5.2 Arithmetic operators

Arithmetic operators are used with numerical values and return a single numerical value as their result.

Operator	Meaning	Example
+	Addition. Adds two <code>number</code> operands. If one of the operands is of type <code>string</code> or is <code>stringlike</code> , the + sign acts as a concatenation operator (see <a href="#">page 72</a> and following example).	4+x
++	Increment. This single-position operator increases the value of an operand by 1. The operand must be a variable with a value which is either of type <code>number</code> or which can be converted to this type. The increased value is assigned to the operand. If the operator is located <b>before</b> the operand, the incremented value is returned. In contrast, if the operator is located <b>after</b> the operand, the original value is returned before incrementation is performed.	x++ (results in 3 if x was originally equal to 3; new value of x is 4) ++x (results in 4 if x was originally equal to 3; new value of x is 4)
-	Subtraction or single-position minus: - If the minus sign is located between two operands then the second operand is subtracted from the first. - As a single-position minus operator, the minus sign is located before the operand. The operand is negated, i.e. the sign is inverted.	two-position: y-4  single-position: -x (-x results in -3 if x equals 3; x keeps the value 3)
--	Decrement. This single-position operator reduces the value of an operand by 1. The operand must be a variable with a value which is either of type <code>number</code> or which can be converted to this type. The reduced value is assigned to the operand. If the operator is located <b>before</b> the operand, the decremented value is returned. In contrast, if the operator is located <b>after</b> the operand, the original value is returned before decrementation is performed.	x-- (results in 3 if x was originally equal to 3; new value of x is 2) --x (results in 2 if x was originally equal to 3; new value of x is 2)
*	Multiplication. Multiplies the two operands.	4*x
/	Division. Divides the two operands.	17 / 4 (results in 4.25)
%	Modulus. Returns the integer remainder of the division of the two operands.	17%4 (results in 1)

The operators `*`, `/` and `%` always return the result at the greatest possible level of precision. Even operations involving whole numbers may yield floating-point results (for example, `17/4` gives the result `4.25`).

If one of the operands in an arithmetic operation is `NaN` (Not a Number), then the result is always `NaN`.

## 5.3 Comparison operators

A comparison operator compares the associated operands and results in a logical value: `true` if the comparison is correct; otherwise `false`.

Operator	Meaning	Example
<code>==</code>	equal to; results in the value <code>true</code> if the operands are equal	<code>3 == 3</code>
<code>!=</code>	not equal to; results in the value <code>true</code> if the operands are not equal	<code>3 != 4</code>
<code>&gt;</code>	greater than; results in the value <code>true</code> if the left-hand operand is greater than the right-hand operand	<code>4 &gt; 3</code>
<code>&gt;=</code>	greater than or equal to; results in the value <code>true</code> if the left-hand operand is greater than or equal to the right-hand operand	<code>4 &gt;= 4</code>
<code>&lt;</code>	smaller than; results in the value <code>true</code> if the left-hand operand is smaller than the right-hand operand	<code>3 &lt; 4</code>
<code>&lt;=</code>	smaller than or equal to; results in the value <code>true</code> if the left-hand operand is smaller than or equal to the right-hand operand	<code>3 &lt;= 4</code>



The numerical value `NaN` does not form part of the linear numberline. A comparison in which one of the operands is `NaN` always returns the value `false`. This is also true in the case of `NaN==NaN`.

### Evaluation of the relational comparison operators (`>`, `>=`, `<`, `<=`)

If both operands are stringlike (see [section “Stringlike data types” on page 47](#)), then both are converted into strings and the result of the lexicographic comparison of these two operands is returned. If one of the operands is undefined or the `null` object, then the result is `false`. Otherwise, both operands are converted to type `number` and the result of the numerical comparison of these two operands is returned.

### Evaluation of the equivalence comparison operators (==, !=)

If both operands are of type `object` or `function`, then the comparison tests whether both operands reference the same object. If one of the operands is the `null` object, then the other operand is converted to type `object` and a comparison is performed.

If one operand is a string and the other stringlike (see [section “Stringlike data types” on page 47](#)), then both operands are converted to type `string` and the result of the comparison is returned.

In all other cases, the two operands are converted to type `number` and a numerical comparison is performed.

### Comparison operators which force a numerical comparison (only in WTML tags)

The operators `#==`, `#!=`, `#>`, `#<`, `#>=`, and `#<=` convert the operands to the numerical data type and return the result of the corresponding numerical comparison. These operators are supported in order to ensure compatibility with WebTransactions V1.x, and are **only** permitted in the conditions of the WTML tags `<wtIf ...>`, `<wtDoWhile ...>`, and `<wtUntil ...>`. They are **not** permitted at any other location in the template.

#### Examples

```
"7" > "10"; //Returns true
"7" > 10;    //Returns false
"7" #> "10" //Returns false (only permitted in WTML tag conditions)
```

## 5.4 Bitwise operators

Bitwise operators treat their operands as sequences of bits (zeros and ones). For example, the decimal number 9 is represented by the bit sequence 1001.

Although bitwise operators transform bit sequences, they return the result as a normal numerical value.

There are bitwise logical operators and bitwise shift operators. It must be possible to convert all the operands to type `number`.

### 5.4.1 Bitwise logical operators (&, |, ^, ~)

Bitwise logical operators function as follows:

- The operands are converted into 32-bit numbers.
- The bits in the two operands are compared pair by pair: the first bit of the left-hand operand corresponds to the first bit of the right-hand operand, the second bit to the second bit, etc.
- The operand is applied to each of these bit pairs and the bitwise result is constructed from the individual partial results.

An exception here is the bitwise NOT operator (~) which is the only one to be single-position, i.e. to take only a single operand. This operator inverts the bits in the operand, i.e. 0 becomes 1 and 1 becomes 0.

The table below illustrates the mode of operation of the bitwise logical operators:

Operator	Description	Example
&	bitwise AND Returns 1 for all bit pairs in which both bits have the value 1.	15 & 9 results in 9 (1111 & 1001 = 1001)
	bitwise OR (inclusive) Returns 1 for all bit pairs in which one or both bits have the value 1.	15   9 results in 15 (1111   1001 = 1111)
^	bitwise XOR (exclusive) Returns 1 for all bit pairs in which exactly one bit has the value 1.	15 ^ 9 results in 6 (1111 & 1001 = 0110)
~	bitwise NOT (complement) Inverts each bit in the operand.	~15 results in -16 (~00...001111 = 11...110000)

## 5.4.2 Bitwise shift operators (<<, >>, >>>)

In all cases, the left-hand operand represents the starting value and the right-hand operand specifies the number of positions to be shifted. Shift operators convert their operands into 32-bit numbers and return a value of type `number`.

Operator	Description	Example
<<	Left shift. The binary representation of the first operand is shifted to the left by the number of positions specified in the second operand. Bits shifted beyond the left-hand edge are ignored and vacant bits at the right-hand edge are filled with the value 0.	9 << 2 results in 36 (1001 by two bits to the left: 100100)
>>	Right shift, taking account of the sign. The binary representation of the first operand is shifted to the right by the number of positions specified in the second operand. Vacant bits at the left-hand edge are filled with the value of the sign (0 for +, 1 for -). Bits shifted beyond the right-hand edge are ignored.	9 >> 2 results in 2 (1001 by two bits to the right: 10)  -9 >> 2 results in -3
>>>	Right shift, ignoring the sign. The binary representation of the first operand is shifted to the right by the number of positions specified in the second operand. Bits shifted beyond the right-hand edge are ignored. Vacant bits at the left-hand edge are filled with the value 0. In the case of non-negative numbers, the operator >>> gives the same result as the operator >>.	19 >>> 2 results in 4 (10011 by two bits to the right: 100)  -9 >>> 2 results in 1073741821

## 5.5 Boolean operators (&&, ||, !)

When boolean (= logical) operators are used, the first operand (in the case of a logical NOT, this is the only operand) is evaluated and converted to type `boolean` if necessary. The return value is either this boolean value or the value of an operand:

Operator	Description	Example
<code>&amp;&amp;</code>	Logical AND. Returns <code>false</code> if the evaluation of the left-hand operand, after conversion to type <code>boolean</code> , results in the value <code>false</code> . In this case, the right-hand operand is not evaluated. Otherwise, the value of the right-hand operand is returned.	<code>"boy" &amp;&amp; "girl"</code> results in: <code>"girl"</code>  <code>3==4 &amp;&amp; "girl"</code> results in: <code>false</code>
<code>  </code>	Logical OR. Returns the value of the left-hand operand if this is evaluated as <code>true</code> . In this case, the right-hand operand is not evaluated. Otherwise, the value of the right-hand operand is returned.	<code>"boy"    "girl"</code> results in: <code>true</code>  <code>3==4    "girl"</code> results in: <code>"girl"</code>
<code>!</code>	Logical NOT. Returns <code>true</code> if the evaluation of the operand results in the value <code>false</code> ; otherwise <code>false</code> .	<code>!"girl"</code> results in: <code>false</code> <code>!(3==4)</code> results in: <code>true</code>

### Truncated evaluation

Boolean expressions are evaluated from left to right. As soon as the result is known, evaluation is aborted.

- `false && anything`  
In the case of a logical AND, if the evaluation of the first operand gives the value `false`, then the second operand is not evaluated.
- `true || anything`  
In the case of a logical OR, if the evaluation of the first operand gives the value `true`, then the second operand is not evaluated.

In the examples above, the expression *anything* is not evaluated. Any possible side effects, for example assignments within *anything*, are therefore not considered.

#### Example

```
x=0;
document.write( false && (x=99) ); //Output: false
document.write(x);                //Output: 0
```

## 5.6 Assignment operators

An assignment operator is located between two operands. It assigns the left-hand operand a value which is based on that of the right-hand operand.

### Equals sign

The most basic assignment operator is the equals sign (simple assignment):

---

```
operand1 =operand2
```

---

The right hand operand is evaluated and the result is assigned to the left-hand operand. The assignment expression itself represents the value. For example, the expression `x=y+1` assigns the value of `y + 1` to `x` and itself represents the value `y+1`. In the expression `z=(x=y+1)`, this value is assigned to the variable `z`.

If the result of the right-hand operand is of type `object` or `function`, a reference is assigned. In all other cases, a value is assigned.

Please note that the type of the left-hand operand may be modified by an assignment operation.

### Example

```
x=7;           // x is of type number
x="otto";     // x is now of type string
x=[1,2,3];    // x is now an array with three elements 1, 2, and 3
x={y:1, z:23}; // x is now an object with the attributes y and z
```

### Assignment operators for standard operations

All the other assignment operators are abbreviated forms of standard operations, as the tables below illustrate:

Operator	Meaning
<code>x += y</code>	<code>x = x + y</code>
<code>x -= y</code>	<code>x = x - y</code>
<code>x *= y</code>	<code>x = x * y</code>
<code>x /= y</code>	<code>x = x / y</code>
<code>x %= y</code>	<code>x = x % y</code>

Assignment operators for arithmetical operations

Operator	Meaning
<code>x &lt;&lt;= y</code>	<code>x = x &lt;&lt; y</code>
<code>x &gt;&gt;= y</code>	<code>x = x &gt;&gt; y</code>
<code>x &gt;&gt;&gt;= y</code>	<code>x = x &gt;&gt;&gt; y</code>
<code>x &amp;= y</code>	<code>x = x &amp; y</code>
<code>x ^= y</code>	<code>x = x ^ y</code>
<code>x  = y</code>	<code>x = x   y</code>

Assignment operators for bitwise operations

## 5.7 String concatenation operator (+)

Alongside the comparison operators, which can also be used with string operands, there is also the concatenation operator `+` which applies solely to strings:

---

*string1 + string2*

---

The `+` operator concatenates the values of the two operands: The result is a **single** string value.

For example, the operation `"good " + "morning"` results in the string `"good morning"`.

A precondition is that at least one of the two operands is **stringlike**. The data types `string` and `function` as well as all objects which possess no `valueOf` method or whose `valueOf` method returns a string are considered to be stringlike.

The abbreviated assignment operator `+=` can also be used with string operands:

If, for example, `mystring` has the value `"good "` then `mystring += "evening"` results in the string `"good evening"`.

*Example: Arithmetical addition compared to string concatenation*

```
document.writeln(4 + 4);           //Output: 8
document.writeln(4 + "4");        //Output: 44
myString=new String(4);
document.writeln(4 + myString); //Output: 44
```

The object `myString` is stringlike since its `valueOf` method returns a string.



## 5.8 Special operators

### 5.8.1 Condition operator (?:)

The condition operator is frequently used as a fast alternative to a simple IF statement. It is the only JavaScript operator to have three operands:

---

```
condition ? expression1 : expression2
```

---

*condition* is evaluated. If, when converted to the type `boolean`, *condition* has the value `true`, then the condition operator returns the value of *expression1*; otherwise, it returns the value of *expression2*. In either case, the other expression is not evaluated (see Example 2 below).

#### Example 1

```
document.write( age >= 30 ? "you're a senior!" : "you're a junior!" );
```

#### Example 2

```
document.write( true ? x="Peter" : y="Paul"); //Output: Peter
document.write( typeof y ); //Output: undefined
```

#### Example 3

This example generates a drop-down list. The condition operator generates a default: for example, if the `COUNTRY` attribute currently has the value 2 then “USA” is displayed by default in the browser.

The evaluation operator `##...#` ensures that the current value of the condition expression (“SELECTED” or “”) is immediately effective in the HTML output.

```
<SELECT Name="COUNTRY" Size=1>
  <OPTION ##host.COUNTRY.Value == 1 ? "SELECTED" : ""# Value="1">Belgium
  <OPTION ##host.COUNTRY.Value == 2 ? "SELECTED" : ""# Value="2">USA
  <OPTION ##host.COUNTRY.Value == 3 ? "SELECTED" : ""# Value="3">Germany
</SELECT>
```

## 5.8.2 Comma operator ( , )

The comma operator first evaluates the left-hand and then the right-hand operands, and returns the value of the right-hand operand:

---

*expression1* , *expression2*

---

It allows you to perform multiple separate evaluations within a single expression. It is frequently used in `for` loops (see Example 2) or in cases where a number of different operations have to be performed within an evaluation operator (see Example 3).

### *Example 1*

```
x=((y=5),4)
```

Although this example is of little practical relevance, it does demonstrate the principle: following the assignment, `x` has the value 4 and `y` the value 5.

### *Example 2*

The comma operator is used in the last `for` loop of the example (the other loops simply create a two-dimensional array and assign values).

It makes it possible to run another variable alongside the loop counter `i` within the conditions of the `for` loop.

```
d=new Array(10);
for (i=0;i<=9;i++) d[i]=new Array(9);
for (i=0;i<=9;i++) {
    for (j=0;j<=9;j++) {
        d[i][j]= i + ":" + j;
    }
}

for (i=0,j=9 ; i<=9 ; i++,j--)
    document.write(d[i][j] + " ; ");
```

This example outputs the values of the “diagonal” array elements:

```
0:9 ; 1:8 ; 2:7 ; 3:6 ; 4:5 ; 5:4 ; 6:3 ; 7:2 ; 8:1 ; 9:0 ;
```

*Example 3*

```
##a=1,b=42,...,""#
```

All comma-separated operations are performed. If an empty string is entered as the last operand of the comma operator, then no output is generated. Within an HTML area, the output of this evaluation operator would therefore be “invisible”.

### 5.8.3 new operator

The `new` operator allows you to create object instances of predefined and user-defined classes.

---

```
objectname = new objecttype([parameters] ...)
```

---

*objectname*

Name of the new object instance

*objecttype*

Object type. This is the name of the associated constructor function. In server-side scripts, the following object types are currently possible:

```
{Object|Boolean|Date|Document|Number|string|Array|RegExp|
WT_Communication|WT_Userexit|user-defined objects}
```

*[parameters]* ...

When you call the constructor functions, you can specify parameters and thereby assign values to the attributes of the new object. The parameters which are actually specified depends on the constructor function in question. Further details can be found in [chapter “Built-in classes and methods” on page 117ff.](#)

*Example*

```
myarray = new Array(20);
```

This expression creates an array object with the name `myarray`, whose first (and at this point whose only) element is assigned the value 20.<sup>1</sup>

---

<sup>1</sup> Up to WTML version 2.0 this method was used to create an array with 20 elements.

### 5.8.4 delete operator

The `delete` operator deletes an object, an object attribute or an array element and releases the reserved memory.

The operator returns `undefined`.

### 5.8.5 in operator

The `in` operator returns a boolean value which indicates whether a particular attribute is contained in a specified object.

---

*attributeNameOrIndex* in *object*

---

*attributeNameOrIndex*

String or numerical expression which represents the name of the attribute or an array index

*objectname*

Name of the object to be examined for the attribute or array index specified in *attributeNameOrIndex*

*Example*

```
a = new Object();
a.b = "abc";
if ("b" in a )      // Returns true
...
if ("c" in a )      // Returns false
```

## 5.8.6 instanceof operator

The `instanceof` operator returns a boolean value which indicates whether a particular object is derived from a specified class.

---

*objectname* instanceof *objecttype*

---

*objectname*

Object to be examined to ascertain whether or not it has been derived from the built-in class *objecttype*

*objecttype*

Class for which the object *objectname* is to be examined

*Example*

```
a= new String ("abc");
b = "abc";
if ( a instanceof String ) . . . // Returns true
if ( b instanceof String ) . . . // Returns false, since specified object
                                // is not a string object
```

## 5.8.7 WT\_THIS (for class templates only)

Within a class template, this keyword returns a reference to the calling host data object. This makes it possible to access the calling host data object in the class template.

For more information on class templates and `WT_THIS`, refer to [chapter “Class templates \(\\*.clt\)” on page 309](#).

## 5.8.8 this

This keyword returns a reference to the calling object within a constructor or method.

*Example*

```
// Method for new class "Employee":
function gibName() {
    return (this.name);
}

// Constructor for class "Employee":
function Employee() {
    // Definition of class attributes:
```

```
this.name      = "";  
this.division  = "development";  
this.machine   = "computer";  
this.worktime  = 35;  
  
// Reference to method:  
this.gibName  = gibName;  
}
```

Outside constructors and methods, `this` returns the global object which contains all globally defined variables.



The objects `WT_SYSTEM`, `WT_POSTED`, `WT_HOST` and globally defined variables that have been created by modules are not attributes of the global object and are thus not returned as attributes of `this`.

#### *Example*

```
a=1;  
b=2;  
c=3;  
for(i in this)  
  document.writeln(i, ' ', this[i]);
```

returns the following output:

```
a: 1  
b: 2  
c: 3
```

You can also use the `this` literal to access global variables within a function that are obscured by local variables of the same name.

#### *Example*

```
a=6;  
function f(x)  
{  
  var a=7;  
  ..return this.a*x;  
}  
res=f(7);
```

In `res`, `42` is returned as usual. `this.a` is used to reference a global variable `a` although a local variable exists with the same name.

## 5.8.9 Evaluation operator ##...#

The evaluation operator evaluates the expression it contains and returns the result as a string. Only exception: if the result is `undefined`, then an empty string rather than the string `"undefined"` is returned.

---

```
## expression #
```

---

*expression* Any expression.

The evaluation operator allows you, for example, to access the current values of objects or object attributes in the template. When you do this, the evaluation operators are replaced by the current values, i.e. they are used in a similar way to a variable.

However, evaluation operators can also be used in contexts in which it is not possible to work with variables:

- in fixed HTML text
- within HTML tags and many WTML tags in order, for example, to set tag properties dynamically. In this case, the evaluation operator may even be located within the string delimiter `" "` or `' '`. Such strings containing evaluation operators are also known as **simple string expressions**.

The evaluation operator is not permitted within `OnCreate` and `OnReceive` scripts. However, you can instead use the `toString` method of the string in question since this offers comparable functionality.

### Examples

```
##WT_SYSTEM.BASEDIR# //Returns the value of the system object attribute
//BASEDIR

##++index# //Returns the index of the template object
//incremented by 1

##void a+=6*7# //Returns an empty string

##WT_HOST.STD.SELECT# //Returns the evaluation of the class template for
//the host data object SELECT
```

See also [“Example 3” on page 73](#).

## Objects in the evaluation operator

The `toString` method is always used for an object which exists only within an evaluation operator. For example, `##hostobject#` has the same meaning as `##hostobject.toString()#`. Thus, in the case of host data objects, the corresponding class template is executed if necessary (see [chapter “Class templates \(\\*.clt\)” on page 309ff](#)).

It is also possible, for example, to write `##WT_SYSTEM#`. In this case, the result is a list of all `WT_SYSTEM` attributes and their values (see [section “toString method” on page 175](#)).<sup>1</sup>

### 5.8.10 typeof operator

The `typeof` operator determines the type of the operand and returns the result as a string. Possible values are: undefined, object, function, number, boolean, or string.

---

`typeof operand`

---

#### *Example*

```
myArray=new Array("Peter", 49, false, null);
document.writeln("type of myArray is: " + typeof myArray + "<BR>");
document.writeln("type of myarray is: " + typeof myarray+"<BR>");
document.writeln("type of myArray.length is: "
    + typeof myArray.length+"<BR>");

for (i in myArray)
    document.writeln("type of " + myArray[i] + " is: "
        + typeof myArray[i]+"<BR>");
```

The example generates the following output:

```
type of myArray is: object
type of myarray is: undefined
type of myArray.length is: number
type of Peter is: string
type of 49 is: number
type of false is: boolean
type of null is: object
```

---

<sup>1</sup> Up to WTML Version 2.0 the result was the same as the result of the `toString` method, namely `[object Object]`.



### 5.8.11 void operator

The `void` operator evaluates the operands but does not return the result of the evaluation. It does not return a “genuine” value but instead the value `undefined`.

---

`void operand`

---

#### *Example*

```
document.writeln("the 'value' of the void expression is: " + void(x=4));  
document.writeln("the value of x is: " + x);
```

The example generates the following output:

```
the 'value' of the void expression is: undefined  
the value of x is: 4
```

## 5.9 Evaluation sequence

The individual operators are distributed over 16 levels of priority:

- operators of the same level are processed in sequence (normally from left to right)
- if the operators belong to different levels then the highest-level operator is processed first

The following table shows the 16 priority levels, starting with the highest priority and ending with the lowest:

Operator type	Individual operators
Call, access, brackets	() on function calls [] e.g.: myArray[2+3] . e.g.: myArray.length
Single-position operators	! ~ - ++ -- typeof void
Multiplication, division, modulus	* / %
Addition/string concatenation, subtraction	+ -
Bitwise shift operators	<< >> >>>
Relational comparison operators	< <= > >=
Equality/inequality	== !=
Bitwise AND	&
Bitwise XOR	^
Bitwise OR	
Logical AND	&&
Logical OR	
Condition operator	?:
Assignment operators	= += -= *= /= <<= >>= >>>= &= ^=  =
Comma operator	,
Evaluation operator	##...#

With the exception of the operators &&, || and ?:, all operands are always evaluated.

### Examples

```
document.writeln( true && true != true && false );//Output: false
x = false ? 1 : 2;                               // x is 2 (not false)
document.writeln(2+3*4);                          //Output: 14 (not 20)
document.writeln(typeof 2+42)                      //Output: number 42 (not number)
```

---

## 6 Global functions

This chapter describes the non-class-specific global functions. For function troubleshooting, see also [section “Exception handling” on page 302](#).

### 6.1 copyFile() function

The global function `copyFile()` copies an existing file.

---

`copyFile(source , destination)`

---

*source*

Path and name of the file to be copied.

*destination*

Path and name of the destination file.

Both files must be located within the base directory. *source* and *destination* can be given relative to the base directory or as absolute specifications.

*Example*

```
copyFile("folder/file1", "folder/filenew");
```

## 6.2 createFolder() function

The `createFolder()` function creates the specified folder in the base directory. The folder declaration is always made relative to the base directory. If the parent directories specified in the path are not available, the *parents* parameter (if it gives the `boolean` result `true`) can specify that they should be generated.

---

```
createFolder(foldername[ , parents])
```

---

*foldername*

Name of the folder relative to the base directory.

*parents*

Specifies that all non-existing parent directories should also be created for this path.

### Result

Boolean value that specifies whether the folder has been created or not.

## 6.3 deleteFile() function

The `deleteFile()` function deletes files and folders from the base directory. Empty folders as well as files are deleted. Folders that are not empty are deleted together with their contents if the *recursive* parameter is set (boolean converted `true`).

---

```
deleteFile(filename[, recursive])
```

---

*filename*            Name of the file or folder relative to the base directory.

*recursive*           Specifies that non-empty folders are deleted together with content.

### Result

Boolean value indicating whether file or folder was deleted or not.

### Example

```
deleteFile("Storage", true);
```

deletes the *Storage* in the base directory together with contents.

## 6.4 escape() function

The global `escape()` function converts special characters within an ASCII string into hexadecimal format. All special characters outside the set { 'A'-'Z', 'a'-'z', '0'-'9', '+', '-', '\*', '/', '\_', '@', '.' } are converted into hexadecimal representations in the format `%nn`.

---

`escape(string)`

---

*string* ASCII string

### Result

The string transferred as an argument, in which all special characters have been converted to hexadecimal format

### Example

```
document.writeln("<BR>" + escape("The_rain. In Spain, Ma'am!"));
```

This example gives the following output:

```
The_rain.%20In%20Spain%2C%20Ma%27am%21
```

### See also

[“unescape\(\) function” on page 114.](#)

## 6.5 eval() function

This function checks the string specified as an argument. If this proves to be one or more valid WTSript statements, the string is executed. If it is an expression, the result is calculated and returned.

The `eval()` function can also be used to dynamically generate WTSript statements or arithmetic expressions in the form of strings, which are then executed or evaluated.

---

`eval(string)`

---

*string* String containing either WTSripts statement or an expression

### Result

*string* is converted to a character string, which is then executed or evaluated as a WTSript program or an expression.

### Example

```
document.writeln("<BR>" + eval("3+7"));           // Expression

// WTSript statement:
x=39;
y=2;
eval("if((x+y+1) == 42) abc='yes'; else abc='no';");
document.writeln("<BR>" + abc);
```

This example gives the following output:

```
10
yes
```

## 6.6 evaluate() function

This function calls the specified template as a type of subroutine and returns the generated HTML text as a string. You can then process this string as required.

The `evaluate()` function acts in the same way as the `include()` function, except for the fact that output that is usually sent to the HTML output stream and then output in the browser is written to a string. In particular, `OnReceive` sections located in the evaluated template are not executed until the next `Receive` point.



If you call `evaluate()` within a function, the same considerations apply as described in [“Notes regarding the use of the include\(\) function within a function” on page 100](#).

---

`evaluate(template)`

---

### *template*

String with the name of a template. *template* is a relative file name. You do not need to specify the file name suffix `.htm` for this file name. WebTransactions searches for the corresponding template on the basis of the set language and style.

The usual search sequence for templates applies here. For more information, refer to the WebTransactions manual “Concepts and Functions”.

### **Result**

The contents of the template are executed and the result is returned as a string. The string is then available for further processing for queries or calculations.

### *Example*

**Template** `test.htm`:

```
<H4> This is a test by USER </H4>
<wtoncreatescript>
<!--
stringInTest = "Hello";
/-->
</wtoncreatescript>
```



The calling template contains:

```
<wtoncreatescript>
<!--
document.write("<br><br><H2>*****evaluate Test*****</H2><br>");
str = evaluate("test");
//Modify and output string
newStr = str.replace("by USER","by the WebTransactions team");
document.write(newStr);
// Variables defined in "test" can now be accessed
document.write(stringInTest);
//-->
</wtoncreatescript>

<H1>End of test template</H1>
```

The example generates the following output:

**\*\*\*\*\*evaluate Test\*\*\*\*\***

This is a test by the WebTransactions team

Hello

End of test template

**See also**

[“include\(\) function” on page 99.](#)

## 6.7 exitDialogStep() function

The global function `exitDialogStep()` terminates the processing of all the templates involved in this dialog step.

Dialog control is described in detail in the WebTransactions manual “Concepts and Functions”.

---

`exitDialogStep()`

---

### *Example*

**Template** test1.htm:

```
<wtOnCreateScript>
<!--
    document.write("before exitDialogStep<BR>");
    exitDialogStep();
    document.write("after exitDialogStep<BR>");
//-->
</wtOnCreateScript>
```

The calling template contains:

```
<wtOnCreateScript>
<!--
document.write("<br><br><H2>*****exitDialogStep Test*****</H2><br>");
include("test1");
document.write("Output of calling template<br>");
//-->
</wtOnCreateScript>
```

The example generates the following output:

**\*\*\*\*\*exitDialogStep Test\*\*\*\*\***

before exitDialogStep

Comment:

In this case, `exitDialogStep()` terminates both the included and the calling template.

### **See also**

[“exitReceiveProcessing\(\) function” on page 91](#), [“exitScript\(\) function” on page 92](#), [“exitSession\(\) function” on page 94](#), and [“exitTemplate\(\) function” on page 95](#).

## 6.8 exitReceiveProcessing() function

This function is valid only in a `ReceiveScript` area. It terminates all current and subsequent `Receive` rules, and does not return any result.

---

```
exitReceiveProcessing()
```

---

### See also

[“exitDialogStep\(\) function” on page 90](#), [“exitScript\(\) function” on page 92](#), [“exitSession\(\) function” on page 94](#), and [“exitTemplate\(\) function” on page 95](#).

## 6.9 exitScript() function

The global function `exitScript()` terminates the processing of the current script area. Processing then continues with the first statement after the script area

---

```
exitScript()
```

---

### *Example 1*

**Template** test1.htm:

```
<wtOnCreateScript>
<!--
    document.write("before exitScript<BR>");
    exitScript();
    document.write("after exitScript<BR>");
//-->
</wtOnCreateScript>
```

The calling template contains:

```
<wtoncreatescript>
<!--
document.write("<br><br><H2>*****exitScript Test*****</H2><br>");
include("test1");
document.write("Output of calling template<br>");
//-->
</wtoncreatescript>
```

The example generates the following output:

**\*\*\*\*\*exitScript Test\*\*\*\*\***

before ExitScript

*Example 2*

```
<wtOnCreateScript>
<!--
document.write ("Before exitScript(<br>");
exitScript();
document.write ("After exitScript(<br>");
//-->
</wtOnCreateScript>
<wtOnCreateScript>
<!--
document.write ("New script area<br>");
//-->
</wtOnCreateScript>
```

This example generates the following output:

```
Before exitScript()
New script area
```

**See also**

[“exitDialogStep\(\) function” on page 90](#), [“exitReceiveProcessing\(\) function” on page 91](#), and [“exitTemplate\(\) function” on page 95](#).

## 6.10 exitSession() function

This function terminates the current WebTransactions session at the earliest possible opportunity. This depends on the location of the `exitSession()` function call:

- When called within a `wtOnCreateScript`, the result of the current WTML document is the last page sent to the browser.
- When called within a `wtOnReceiveScript`, the WebTransactions session is closed following generation of the next synchronized output.

This function behaves in the same way as `WT_SYSTEM.EXIT_SESSION="TRUE"`.

---

`exitSession()`

---



`exitSession()` terminates processing at the current position and closes the WebTransactions session after this dialog step. If you want to terminate the session immediately, you must also call the `exitDialogStep()` function. Otherwise, the statements following `exitSession()` will still be executed before the session is terminated.

### See also

Global system object attribute `PREVENT_EXIT_SESSION` (see the WebTransactions manual "Concepts and Functions"), [“exitDialogStep\(\) function” on page 90](#), [“exitReceiveProcessing\(\) function” on page 91](#), [“exitScript\(\) function” on page 92](#) and [“exitTemplate\(\) function” on page 95](#).

## 6.11 exitTemplate() function

This function terminates processing of the current template. Processing continues with the next statement in the calling template. This can be thought of as a return from a subroutine.

The result of an `exitTemplate()` function call at the top template level is the same as that of an `exitDialogStep()` function call.

---

`exitTemplate()`

---

### *Example*

**Template** test1.htm:

```
<wtOnCreateScript>
<!--
    document.write("before exitTemplate<BR>");
    exitTemplate();
    document.write("after exitTemplate<BR>");
//-->
</wtOnCreateScript>
```

The calling template contains:

```
<wtOnCreateScript>
<!--
document.write("<br><br><H2>*****exitTemplate Test*****</H2><br>");
include("test1");
document.write("Output of calling template<br>");
//-->
</wtOnCreateScript>
```

The example generates the following output:

**\*\*\*\*\*exitTemplate Test\*\*\*\*\***

before exitTemplate

Output of calling template

### *Comment*

This terminates only the processing of the included template. Processing of the calling template continues.

### **See also**

[“exitDialogStep\(\) function” on page 90](#), [“exitReceiveProcessing\(\) function” on page 91](#), [“exitScript\(\) function” on page 92](#), and [“exitSession\(\) function” on page 94](#).

## 6.12 forward() function

This function searches for the template specified in the argument and starts to process it. Control is passed to this template and is not returned to the calling template. This means that all the statements in the calling template that follow the `forward()` call are not executed. Any `Receive` rules that have been read up to this point are executed on the next `Receive`.

If the specified template is not found, then an error message is output and the WTScrip script continues after `forward()`.



When you call `forward()` within a function, the same considerations apply as described in [“Notes regarding the use of the include\(\) function within a function” on page 100](#).

---

`forward(template)`

---

*template*

String with the name of a template. *template* is a relative file name. You do not need to specify the file name suffix `.htm` for this file name. WebTransactions searches for the corresponding template on the basis of the set language and style

The usual search sequence for templates applies here. For more information, refer to the WebTransactions manual “Concepts and Functions”.

### Result

Processing control is passed to the specified template.

*Example*

Template `test.htm`:

```
<H4> This is a test by USER </H4>
<wtoncreatescript>
<!--
stringInTest = "Hello";
//-->
</wtoncreatescript>
```



The calling template contains:

```
<wtoncreatescript>
<!--
document.write("<br><br><H2>*****forward Test*****</H2><br>");
forward("test");
document.write("This text is displayed only if an error occurs");
//-->
</wtoncreatescript>
```

The example generates the following output:

**\*\*\*\*\*forward Test\*\*\*\*\***

This is a test by USER

### See also

[“evaluate\(\) function” on page 88](#) and [“include\(\) function” on page 99](#).

## 6.13 import function()

The `import()` function enables you to load a template as a module. Functions and variables that you define in modules are available throughout the entire WebTransactions session.

You can find more information about modules and the procedure to follow to load a template in the WebTransactions manual “Concepts and Functions”, section ‘Master, class and module templates’.

---

`import(template)`

---

*template*

Character string with the name of a template. *template* is a relative file name. You do not need to specify the file name suffix `.htm` for this file name. WebTransactions searches for the corresponding template on the basis of the set language and style.

The template is found using the search sequence of WebTransactions; see also the WebTransactions manual “Concepts and Functions”.

The folder `<WebTA Install Directory>/modules` is added to the search path for templates in the last position in order to enable explicit loading of optional standard modules.

## 6.14 include() function

This function calls the specified template as a type of subroutine. The results of the included template are output directly in the HTML output stream.

The `include()` function can also be used to include templates in WTSript areas.

---

`include(template)`

---

### *template*

String with the name of a template. *template* is a relative file name. You do not need to specify the file name suffix `.htm` for this file name. WebTransactions searches for the corresponding template on the basis of the set language and style

The usual search sequence for templates applies here. For more information, refer to the WebTransactions manual "Concepts and Functions".

### Result

The included template is inserted in full. Within the included template, you can use the same language elements as in any other template, including `wtOnCreate` scripts and `wtOnReceive` scripts. However, the WTML tags must be syntactically complete in each template, e.g. it is *not* permissible to start an IF control structure in the including template and then close it in the included template.

### *Example*

Template `test.htm`:

```
<H4> This is a test by USER </H4>
<wtoncreatescript>
<!--
stringInTest = "Hello";
/-->
</wtoncreatescript>
```

The calling template contains:

```
<wtoncreatescript>
<!--
document.write("<br><br><H2>*****Include Test*****</H2><br>");
include("test");
// Variables defined in "test" can now be accessed
document.write(stringInTest);
//-->
</wtoncreatescript>

<H1>End of test template</H1>
```

The example generates the following output:

**\*\*\*\*\*Include Test\*\*\*\*\***

This is a test by USER

Hello

End of test template

### See also

[“evaluate\(\) function” on page 88](#), [“forward\(\) function” on page 96](#) and [“import function\(\)” on page 98](#).

### Notes regarding the use of the include() function within a function

If you call `include()` (or `forward()` and `evaluate()`) within a function, functions and variables are available within this function which are defined in the included template using the `var` keyword. These are then local variables and functions.

Functions from the included template can call each other. Constructors and method definitions in the included templates can also be executed locally.

If a constructor from the included template is used to create an object that exists for longer than the function in which it was defined, then the following problem may occur: Global help functions defined in the included template and used within the methods are no longer available after the outer function has been completed.

For this reason, you should always define and use these help functions as methods of the object.

*Example*

The `include` function for including the `myClass.htm` template is called within the `outer()` function:

```
function outer()
{
    include('myClass.htm');
    return new myClass();
}
myObject= outer();
myObject.myMethod();
```

Problems occur when `myClass.htm` defines the class in the following manner:

The `twice()` help function is used in the `myMethod()` function and is no longer available outside the `outer()` function.

```
function myClass()
{
    this.myMethod=myMethod;
}
function myMethod()
{
    document.write( twice(21) );
}
function twice(x)
{
    return 2*x;
}
```

It is recommended that the `twice()` help function is also defined as a method in order to make sure that `twice()` remains available after completion of the `outer()` function.

```
function myClass()
{
    this.myMethod=myMethod;
    this.twice=twice;
}
function myMethod()
{
    document.write( this.twice(21) );
}
function twice(x)
{
    return 2*x;
}
```

## 6.15 isRequestWaiting() function

The global function `isRequestWaiting()` queries whether a new request is waiting for processing (e.g. data posted by the browser).

No new requests can be sent to a WebTransactions process as long as the process is in a loop in a function triggered by `setTimeout()`. `isRequestWaiting()` can be used in a loop of this sort to query whether a new request is waiting.

---

`isRequestWaiting()`

---

### Result

Boolean value specifying whether a new request is waiting for processing.

### Example

```
<html>
<body>
<wtoncreatescript>
  if(typeof WT_SYSTEM.counter != 'number' )
  WT_SYSTEM.counter=0;
  if(WT_POSTED.quit)
  {
    document.write('bye '+WT_SYSTEM.counter);
    exitSession();exitTemplate();
  }
  WT_SYSTEM.ex = new WT_Userexit( "WTSsystemExits" );
  function backgroundLoop()
  {
    do
    {
      WT_SYSTEM.ex.WTSleep(1000);WT_SYSTEM.counter++;
    } while(! isRequestWaiting());
  }
</wtoncreatescript>
<form webtransactions>
##WT_SYSTEM.counter++#
<input type="submit" name="step" value="step">
<input type="submit" name="background" value="background">
<input type="submit" name="quit" value="quit">
<wtOnReceiveScript>
  if( WT_Posted.background )
    setTimeout('backgroundLoop()', 500);
</wtOnReceiveScript>
</form>
```

```
</body>  
</html>
```

## 6.16 listFolder() function

The `listFolder()` function provides the contents list of a folder from the base directory. It returns an array of string objects, where each string object describes a contained file named in correspondence to the optional *pattern* parameter. The folders '.' (dot) and '..' (dotdot) are not housed in the array. Links are not resolved but in Windows the extension '.lnk' is removed from the file name. The value of the string object is the name of the file.

The directories '.' (dot) and '..' (dot dot) are by default not included in the array. The parameter *all* (if it returns `true` after conversion to `boolean`) can be used to specify that file names that start with '.' (dot) are also to be listed.

The object has the following attributes:

*isDir* Of `boolean` type. Specifies whether the described file is a folder.

*size* Of `number` type. Indicates file size in bytes.

*lastAccess*

Of `number` type. Date of last access in milliseconds (see example).

*lastModified*

Of `number` type. Date of last modification in milliseconds (see example).

---

```
listFolder(foldername[, pattern [, all]])
```

---

*foldername*

Name of the folder relative to the base directory. If you wish to list the base directory a slash '/' must be used for the *foldername*.

*pattern*

Pattern to which the foldername must correspond (optional).

The pattern complies with the rules for specifying a partially qualified file name on the platform on which WebTransactions is deployed.

*all* Specifies that file names that start with '.' (dot) are also to be listed (optional).

### Result

Array of string objects. If the folder does not exist, `undefined` is returned.



*Example 1*

Output of all .htm files in config/forms folder:

```
dirList=listFolder("config/forms","*.htm");
for (i=0;i<dirList.length;i++)
document.write(dirList[i]+"<br>");
```

The example generates the following output:

```
AutomaskOSD.htm
example.htm
StartTemplateHTTP.htm
StartTemplateOSD.htm
wtasync.htm
wtBrowserFunctions.htm
wtKeysOSD.htm
wtPkeyFunctions.htm
wtPKEYS.htm
wtPKeyValues.htm
```

*Example 2*

Use of *pattern* and the attributes *lastAccess* and *lastModified* for output as date string:

```
fileArr = listFolder('/config/forms','*.htm');
document.write("Last access on " + fileArr[0] + ": ");
document.writeln((new Date(fileArr[0].lastAccess)).toLocaleString());
document.write("<br>Last modification to " + fileArr[0] + ": ");
document.writeln((new Date(fileArr[0].lastModified)).toLocaleString());
```

The example generates the following output:

```
Last access on AutomaskOSD.htm: 04/02/03 13:22:24
Last modification to AutomaskOSD.htm: 04/02/03 13:22:24
```

*Example 3*

```
listFolder("folder");
listFolder("folder","*.txt");
```

Use of *all* to also list filenames starting with '.' (dot):

```
listFolder("folder","*",true);
```

This example generates the following output:

```
[[new String("file.txt"),(new String("file1"))]
[[new String("file.txt")]
[[new String(".invisible"),(new String("file.txt")), (new String("file1"))]
```

## 6.17 moveFile() function

The global function `moveFile()` moves a file to a different directory in the base directory or renames a file.

---

```
moveFile(name_1, name_2)
```

---

*name\_1*

Path and name of the file to be moved/renamed.

*name\_2*

Path and name of the destination file.

Both files must be located within the base directory. *name\_1* and *name\_2* can be given as relative to the base directory or as absolute specifications.

*Example*

```
moveFile("/folder/filenew", "/folder/filerename");
```

## 6.18 Number() function

The global function `Number()` converts an expression to the data type `number` or, if this is not possible, to `NaN`.

---

`Number(expression)`

---

*expression*

Any expression

### Result

The conversion process is performed as described for the target data type `number` in [section “Type conversion” on page 47](#).

If *expression* cannot be converted to numerical format, `NaN` is returned.

### Example

```
object1="hello world!";
document.writeln("<BR>" + "The value of " + object1 + " is: " +
    Number(object1));
object2=new String("42");
document.writeln("<BR>" + "The value of " + object2 + " is: " +
    Number(object2));
```

This example gives the following output:

```
The value of hello world! is: NaN
The value of 42 is: 42
```

## 6.19 parseFloat() function

This function converts a string to its numeric value or, if the string is not a number, to NaN.

---

`parseFloat(string)`

---

*string* String with numerical contents

### Result

The numerical value of *string* if *string* contains a numerical representation; otherwise, NaN.

If `parseFloat()` detects a character that is not a numerical value in its argument, the value is evaluated up to this point and returned. The rest of the argument is ignored. Thus "22a" gives 22, "2.3.4" gives 2.3.

Leading and trailing spaces are permitted.

### Example

```
string1="hello world!";
document.writeln("<BR>" + "The value of \"" + string1 + "\" is: " +
    parseFloat(string1));
string2=" +3.1415927 ";
document.writeln("<BR>" + "The value of \"" + string2 + "\" is: " +
    parseFloat(string2));
```

This example gives the following output:

```
The value of "hello world!" is: NaN
The value of "+3.1415927" is: 3.1415927
```

## 6.20 parseInt() function

This function converts a string to its numerical value or, if the string is not a number, to NaN. It can take into consideration the number notation used.

---

```
parseInt(string[, base])
```

---

*string* String with numerical contents

*base* Base for the number specified in *string*. This must be an integer greater than 1 but less than or equal to 36. If *base* is not specified, the default value is 10.

### Result

Numerical value of *string* if *string* contains the representation of an integer; otherwise, NaN.

If *string* contains a character other than a digit in the specified base, the value is evaluated up to that character and returned. The rest of the argument is ignored. Leading and following blanks are permitted.

### Example

```
document.writeln("<BR>" + parseInt("F", 16));  
document.writeln("<BR>" + parseInt("1111", 2));  
document.writeln("<BR>" + parseInt("F", 10));  
document.writeln("<BR>" + parseInt("15"));  
document.writeln("<BR>" + parseInt("4.2"));
```

This example gives the following output:

```
15  
15  
NaN  
15  
4
```

## 6.21 setNextPage() function

This function defines the WTML document to be used for the next synchronized dialog step. It sets the `WT_SYSTEM.FORMAT` attribute to the name of the new document.

---

`setNextPage(documentName)`

---

### *documentName*

This argument is converted to a string which is interpreted as a file name. The file name extension `.htm` may be omitted. The specified document is then located and selected in accordance with the system object settings for the style (`WT_SYSTEM.STYLE`) and language (`WT_SYSTEM.LANGUAGE`) and the search strategy defined in `WT_SYSTEM.FORMAT`.

The generally valid search sequence for templates applies. For more information see the WebTransactions manual “Concepts and Functions”.

## 6.22 setSingleStep() function

The function `setSingleStep()` is used in conjunction with the WebLab Offline Single Step Tracking, a utility used when generating pages and running `Receive` scripts to log each line executed together with the associated variables and their values in a file. Once generation is complete,

WebLab can then process this file and track the code sequence.

The `setSingleStep()` function allows you to control the amount of information logged. `setSingleStep("on")` switches the logging function on, while `setSingleStep("off")` switches it off. You can switch the logging function on and off repeatedly within a template by inserting `setSingleStep()` calls at the appropriate locations. In this case, only steps carried out between the `setSingleStep("on")` and `setSingleStep("off")` calls will be recorded.

---

```
setSingleStep( { "on" }
               { "off" } )
```

---

"on"    Activates the logging of single steps

"off"   Deactivates the logging of single steps

### *Example*

In the following example, only `taggedOutput` calls are logged in the `AutomaskOSD`.

```
...
<wtOnCreateScript>
<!--
...
  for (element = OSD_0.$FIRST.Name; OSD_0 && element != '$END';
       element = OSD_0.$NEXT.Name)
  {
    ...
    if ( currentHostObject.Type == 'Protected' &&
        currentHostObject.Markable == 'No' )
    {
      setSingleStep ("on");
      taggedOutput( currentHostObject );
      setSingleStep ("off");
    }
    ...
  }
//-->
</wtOnCreateScript>
```

## 6.23 setTimeout() function

The global `setTimeout()` function is used to execute delayed processing. The script is always handled without interruption (see, however, [“isRequestWaiting\(\) function” on page 102](#)).

The Timeout script is not handled until the specified time has elapsed. During the wait time outside queries, e.g. user browser inputs, can be handled. The start of the Timeout script is, if necessary, further delayed through "simultaneous" user inputs or asynchronous WebTransactions queries to the session. As a rule however, no noticeable delay occurs. In this way it is possible to combine page generation and delayed handling of scripts without unnecessary wait time.

Delayed handling of script parts that have not yet been started is prevented or suppressed by termination of the session.

Handling always takes place in a synchronous context.

---

`setTimeout(script, milliseconds)`

---

*script* Character string that either contains an expression or WTSript statements that have to be executed.

*milliseconds*

Minimum number of milliseconds that must elapse before the execution of *script*.

### Result

None.

### Example

Example of continuous processing at 15-second intervals:

```
<wt0n...Script>
  function doit ()      { // do anything;};
  WT_SYSTEM.doit       = doit;
  WT_SYSTEM.endlessTask = "WT_SYSTEM.doit()";
  WT_SYSTEM.endlessTask += "setTimeout( WT_SYSTEM.endlessTask, 15000 );"
  setTimeout( WT_SYSTEM.endlessTask, 15000 );
</wt0n...Script>
```



In the WTSript statements that you want to execute with a delay, access only functions and variables that are available throughout the entire session and therefore stored, for example, under `WT_SYSTEM`.

If no further dialog steps are received from the browser, the session terminates when `WT_SYSTEM.TIMEOUT_USER` has expired.



## 6.24 setTraceLevel() function

The global function `setTraceLevel()` is used to turn a trace on or off while a session is running. The trace can thus be restricted to certain parts of templates to keep the trace file small and focussed on specific sections of the session.

---

`setTraceLevel(string)`

---

*string* The string "FULL" (full trace activated) or "NONE" (trace deactivated)

### See also

[“writeToTrace\(\) function” on page 115.](#)

## 6.25 String() function

This function converts the result of an expression into string format.

---

`String(expression)`

---

*expression*

Any expression

### Result

A string that represents the object

The conversion process is performed as described for the target data type `string` in [section “Type conversion” on page 47.](#)

### Example

```
value1 = true;
document.writeln("<BR>value1 = " + String(value1));
```

```
timeNow = new Date ();
document.writeln("<BR>The current date and time is " + String(timeNow));
```

This example gives the following output:

```
value1 = true
The current date and time is Mon Jul 05 16:33:06 1999
```

## 6.26 unescape() function

The global function `unescape()` converts hexadecimal representations in the format `%nn` of characters within an ASCII string back into the original characters.

---

`unescape(string)`

---

*string* ASCII string containing characters in hexadecimal format

### Result

The string transferred as the argument, in which all hexadecimal character representations are converted back into the original characters

### Example

```
document.writeln("<BR>" +
    unescape("The_rain.%20In%20Spain%2C%20Ma%27am%21"));
```

This example gives the following output:

```
The_rain. In Spain, Ma'am!
```

### See also

[“escape\(\) function” on page 86.](#)

## 6.27 writeToTrace() function

This function writes an entry in the WebTransactions trace file each time the trace option is activated. This trace entry is preceded by the text `writeToTrace()`: for identification purposes.

---

`writeToTrace(expression)`

---

*expression*

Any expression. This expression is converted to the data type `String` and written to the trace file each time the trace option is activated.

### See also

[“setTraceLevel\(\) function” on page 113.](#)



---

## 7 Built-in classes and methods

Every variable of type `object` belongs to a specific class, which determines the methods which are available for the object. For example, the `replace` method can be used with all objects of the `String` class. For many classes, there are also predefined attributes which are automatically possessed by every object in this class. For instance, all strings have the `length` attribute, which specifies the number of characters in the string.

This chapter describes the built-in classes and the predefined attributes and methods which are supported by WebTransactions in alphabetical order. These classes behave in the same way as the corresponding JavaScript classes.

## 7.1 Array class

Arrays are objects whose attributes are named by means of indices. Indices are non-negative integers beginning with 0.

Arrays are created with a specified length and can be dynamically extended. The individual attributes of an array may be of different types. An array may contain another array as one of its attributes. In this way, you can create multidimensional arrays (see [“Example 2: Two-dimensional array” on page 119](#)).

### 7.1.1 Constructors

---

```
Array()  
Array(comp)  
...  
Array(comp, comp, ...)
```

---

#### Return value

Object of the `Array` class

#### Parameters

*comp* Component of the array

If the constructor is called without an argument, an array of length 0 is created without attributes.

If the constructor is called with a single argument, an array of length 1 is created; an attribute is generated for the argument and the argument is assigned to it.<sup>1</sup>

If the constructor is called with several arguments, an array of the corresponding length is created. An attribute is generated for each argument and assigned to it.



An array can also be created directly by means of assignment in the literal notation (see Example 1 below).

---

<sup>1</sup> Up to WTML version 2.0, this constructor is used to create an argument of the given length.

*Example 1*

```
a = new Array();
b = new Array(25);
c = new Array(4, "Peter");
d = [1, "string1"];
```

*Explanation:*

**a** is an array with no attributes, i.e. of length 0.

**b** is an array of length 1. The attribute `b[0]` is of type `number` and has the value 25.<sup>1</sup>

**c** is an array of length 2. The attribute `c[0]` is of type `number` and has the value 4; `c[1]` is of type `string` and has the value "Peter".

**d** is an array of length 2. The attribute `d[0]` is of type `number` and has the value 1; `d[1]` is of type `string` and has the value "string1".

*Example 2: Two-dimensional array*

```
d=new Array();
for (i=0;i<=2;i++) d[i]=new Array();

for (i=0;i<=2;i++) {
    str=i+1 + ".Row: | ";
    for (j=0;j<=4;j++) {
        d[i][j]=i+"," +j+ " | ";
        str+=d[i][j];
    }
    document.write(str+"<BR>");
}
```

*Explanation:*

The first two lines of this example define a two-dimensional array.

The subsequent nested `for` loops assign values to the array elements (in each case a string containing the indices). The values for each row are combined in the string `str` and output.

This example gives the following output:

```
1.Row: | 0,0 | 0,1 | 0,2 | 0,3 | 0,4 |
2.Row: | 1,0 | 1,1 | 1,2 | 1,3 | 1,4 |
3.Row: | 2,0 | 2,1 | 2,2 | 2,3 | 2,4 |
```

<sup>1</sup> Up to WTML version 2.0, `b` is created as an array with a length of 25.

## 7.1.2 Attributes

---

length

---

The `length` attribute is an integer which specifies the number of elements in the array. An initial length is always defined implicitly by the constructor.

If new indices are assigned and created, then the length of the array is always 1 greater than the largest index used (since the indices start at 0 rather than at 1).

In the case of multi-dimensional arrays, only the elements of the first dimension are taken into account. This is because the array objects of other dimensions are themselves independent objects (see the two-dimensional array `g` in the example below).

### *Example*

```
e=new Array();
document.write("Length of array e is: " + e.length + "<BR>");

f=new Array(25);
document.write("Length of array f is: " + f.length + "<BR>");

f[99]="last element";
document.write("New length of array f is: " + f.length + "<BR>");

g=new Array();
for (i=0;i<=2;i++) g[i]=new Array(6);
document.write("Length of array g is: " + g.length + "<BR>");
```

This example gives the following output:

```
Length of array e is: 0
Length of array f is: 100
New length of array f is: 100
Length of array g is: 3
```



### 7.1.3 concat method

This method joins the calling array with another array transferred in the *array* argument to form a result array. The calling array and the array transferred in *array* remain unchanged

---

```
concat(array)
```

---

#### Return value

An array consisting of the calling array and the array transferred in the *array* argument

#### Parameters

*array* Array which is to be appended to the calling array to form a result array

#### Example

```
h=new Array("Maria","Lena","Hilda");
i=new Array("Frank","Oscar","Harry");
j=h.concat(i);
document.write(j.length + " ");
document.write(j);
```

This example gives the following output:

```
6 ["Maria","Lena","Hilda","Frank","Oscar","Harry"]
```

### 7.1.4 equals method

This method compares the calling array object with the object transferred in the *object* argument for equality of class, attributes, and values.

Two arrays are said to be equal if they have the same number of elements and the same values for corresponding array elements (elements with the same index or the same attribute names in the case of associative arrays).

---

`equals(object)`

---

#### Return value

Boolean return value: `true` if the two objects are equal, `false` if not

#### Parameters

*object* Object with which the calling object is to be compared

#### Example

```
myArray1 = new Array("a","b","c");
myArray2 = new Array("a","b","c");

document.write(myArray1.equals(myArray2)? "Arrays equal": "Arrays unequal");
```

This example gives the following output:

```
Arrays equal
```

### 7.1.5 `getClassName` method

This method returns a string specifying the name of the class to which the calling object belongs.

---

```
getClassName()
```

---

#### **Return value**

String specifying the class to which the calling object belongs, in this case `Array`

#### **Parameters**

None

#### *Example*

```
arr = new Array();  
arrClass = arr.getClassName(); // arrClass is a string containing  
                               // the class name "Array"
```

### 7.1.6 join method

This method converts all the elements of an array into strings and combines them in a single string. Undefined elements are converted into blank strings.

---

```
join()  
join(separator)
```

---

#### Return value

String formed from the elements of the calling array object

#### Parameters

*separator*

String expression which is to be used to separate the individual elements. If *separator* is not already of type `string`, it is converted to this type. If *separator* is not specified, the elements are separated by commas (optional).

#### Example 1

```
h=new Array("Maria","Lena","Harry");  
document.write(h.join(" - ") + ".");
```

This example gives the following output:

Maria - Lena - Harry.

#### Example 2

```
d=new Array();  
for (i=0;i<=2;i++) d[i]=new Array();  
for (i=0;i<=2;i++) {  
    for (j=0;j<=4;j++) {  
        d[i][j]=i+":"+j;  
    }  
}  
document.write(d.join("<BR>"));
```

This example gives the following output:

```
["0:0","0:1","0:2","0:3","0:4"]  
["1:0","1:1","1:2","1:3","1:4"]  
["2:0","2:1","2:2","2:3","2:4"]
```

### 7.1.7 pop method

This method deletes the last array element and returns its value. It modifies the calling object.

---

```
pop()
```

---

#### **Return value**

Value of the last array element

#### **Parameters**

None

*Example*

See [push method](#)

### 7.1.8 push method

This method appends the specified elements *elem1*, *elem2*, etc. to the calling array and returns the new array length. It modifies the calling object.

---

```
push(elem1)
push(elem1, elem2)
...
push(elem1, elem2, ...)
```

---

#### Return value

Length of the array after the `push` call

#### Parameters

*elem1*, *elem2*, ...

Elements to be appended to the array

#### Example

```
j=new Array();
k=j.push("a", "b", "c");
l=j.pop();
document.write(k + "<BR>");
document.write(l + "<BR>");
document.write(j);
```

This example gives the following output:

```
3
c
["a", "b"]
```

### 7.1.9 reverse method

This method reverses the sequence of the array elements, so that the first element becomes the last element and the last element becomes the first. It modifies the calling object.



If the `reverse` method is applied to an object belonging to a class derived from the `Array` class, the inheritance principle is implicitly violated (see example). This is because this method assigns the respective prototype value to the corresponding instance attribute for all array elements before execution.

---

`reverse()`

---

#### Return value

Reference to the calling object

#### Parameters

None

#### Example

```
// "Normal" arrays:
j1=new Array("a","b","c");
k1=j1.reverse();
document.write("k1: " + k1 + "<BR>");
document.write("j1: " + j1 + "<BR>");

// Derived classes:
j2=new Array("a","b","c");
function MyArray() {}
MyArray.prototype = j2;
k2=new MyArray();
k2.reverse();
j2[0]="d";
document.write("k2: " + k2 + "<BR>");
document.write("j2: " + j2);
```

This example gives the following output:

```
k1: ["c","b","a"]
j1: ["c","b","a"]
k2: ["c","b","a"]
j2: ["d","b","c"]
```

### 7.1.10 **shift** method

This method deletes the first array element.

---

```
shift()
```

---

#### **Return value**

Deleted array element

#### **Parameters**

None

#### *Example*

See `unshift` method on [page 135](#).



### 7.1.11 slice method

This method returns the section of the calling array which extends from *index1* to *index2*-1. It does not modify the calling object.

---

```
slice(index1)  
slice(index1, index2)
```

---

#### Return value

Section of the calling array which extends from *index1* to *index2*-1

#### Parameters

*index1* Index of the element in the calling array which is to be the first element of the result array.

If *index1* < 0 or is greater than the length of the array, an empty array is returned.

*index2* Index of the first element in the calling array which is not to be included in the result array.

If *index2* is not specified, the result array extends to the end of the calling array (same as `slice(index1, callingArray.length)`).

If *index2* is less than *index1*, an empty array is returned.

If *index2* < 0, it specifies the offset from the end of the array (same as `slice(index1, callingArray.length+index2-1)`). If this index is smaller than *index1*, an empty array is returned.

#### Example

```
j=new Array("a", "b", "c", "d", "e");  
k=j.slice(1,3);  
l=j.slice(2,-1);  
document.write(j + "<BR>");  
document.write(k + "<BR>");  
document.write(l + "<BR>");
```

This example gives the following output:

```
["a","b","c","d","e"]  
["b","c"]  
["c","d"]
```

## 7.1.12 sort method

This method sorts an array in ascending order. It modifies the calling array. If the method is called without any arguments then the values of all the defined indices are converted into strings and sorted lexicographically (for example, “14” will come before “3”). Undefined array elements are converted into empty strings and entered at the end of the array (see “[Example 2](#)” on page 131).

---

```
sort()  
sort(compareFunction)
```

---

### Return value

Calling array sorted in ascending order

### Parameters

*compareFunction*

If you want to apply a different sort order, then you must specify the name of a function which is defined as follows:

```
function compareFunction(a,b){...return ..}
```

The return value controls the comparison:

- If *a* is to precede *b*, then the function must return a numerical value less than 0.
- If *b* is to precede *a*, then the function must return a numerical value greater than 0.
- If the order in which *a* and *b* occur is of no significance, then the function must return the value 0.

If you want to compare numbers rather than strings, then the comparison function simply needs to subtract *b* from *a* (see “[Example 3](#)” on page 131).



If the `sort` method is applied to an object belonging to a class derived from the `Array` class, the inheritance principle is implicitly violated (see “[Example 5](#)” on page 132). This is because this method assigns the respective prototype value to the corresponding instance attribute for all array elements before execution.

*Example 1*

```
myArray=new Array("Pit","Zoe","Adam");
document.write(myArray.sort());
```

This example gives the following output:

```
["Adam","Pit","Zoe"]
```

*Example 2*

```
a=new Array();
a[0]="Zoe";
a[4]="Adam";
b=a.sort();
document.write(b[1]);
```

This example gives the following output:

```
Zoe
```

*Example 3*

```
function compareNumber(a,b) {
    return a-b;
}

myArray=new Array(14,3,9,"-2",-8);
sortedArray=myArray.sort();
document.write(sortedArray + " (without comparison function) <BR>");
sortedArray=myArray.sort(compareNumber);
document.write(sortedArray + " (with comparison function)");
```

This example gives the following output. The output illustrates that numbers are correctly sorted in numerical order if the comparison function is used - even when numerical strings are being processed ("-2"). This is because the operands of the two-position minus operator are converted to type number.

```
["-2",-8,14,3,9] (without comparison function)
[-8,"-2",3,9,14] (with comparison function)
```

*Example 4*

```
function CompareLength(a,b) {
    if (a.length < b.length)
        return -1;
    if (a.length > b.length)
        return 1;
    return 0;
}
myarray=new Array("Sebastian","Eva","Roberta","Mike","Martin","Ursula");
sorted_array=myarray.sort(CompareLength);
document.write(sorted_array.join("<BR>"));
```

**This example gives the following output:**

```
Eva
Ursula
Mike
Martin
Roberta
Sebastian
```

*Example 5*

```
a = [5, 7, 4];

function MyArray() {}
MyArray.prototype = a;

b = new MyArray();    // b = [5, 7, 4]
a[3] = 1;
a[1] = 9;              // b = [5, 9, 4, 1] (due to inheritance)

b.sort();             // b = [1, 4, 5, 9]

a[1] = 7;             // b = [1, 4, 5, 9] (inheritance violated)
```

### 7.1.13 splice method

This method returns a section of the calling array and if necessary replaces it with the optional elements specified. It modifies the calling object.

---

```
splice(index, count)
splice(index, count, elem1)
splice(index, count, elem1, elem2)
...
splice(index, count, elem1, elem2, ...)
```

---

#### Return value

Array containing the elements removed from the calling array

#### Parameters

*index* Index of the first element to be deleted in the calling array

If *index*<0, *index*=0 is assumed.

*count* Number of elements to be deleted from the calling array.

If *count*=0, no array elements are deleted.

If *count* is greater than the number of remaining elements in the array, all remaining elements are deleted.

If *count* < 0, no elements are deleted.

*elem1, elem2, ...*

(Optional) If at least one element is specified, this is inserted together with all other specified elements at the position in the calling array at which elements were previously deleted.

#### Example

```
j=new Array("a", "b", "c", "d");
document.write(j + "<BR>");
k=j.splice(2,1,"z");
document.write("Deleted: " + k + "!" + "<BR>");
document.write(j + "<BR>");
```

This example gives the following output:

```
["a","b","c","d"]
Deleted: ["c"]!
["a","b","z","d"]
```

### 7.1.14 toString method

This method transforms the calling array into a string containing the values of the array elements separated by commas:

```
["element0", "element1", ...]
```

In order to avoid endless chains, the `toString` method will terminate output in the event of recursion, i.e. output will be stopped as soon as the same object reference is encountered a second time

---

```
toString()
```

---

#### Return value

String containing values of the elements of the calling array, separated by commas

#### Parameters

None

#### Example

```
myArray = new Array("This", "is", "a", "string");
document.write(myArray.toString());
myArray[4] = new String("and an object");
document.write(myArray.toString());
```

This example gives the following output:<sup>1</sup>

```
["This","is","a","string"]
["This","is","a","string",(new String("and an object"))]
```

You will find an example illustrating recursion in the description of `Object.toString()` on [page 175](#).

---

<sup>1</sup> Up to WTML version 2.0, there are no square brackets `[]` in the entry.

### 7.1.15 unshift method

This method inserts the specified elements *elem1*, *elem2*, etc. at the start of the calling array and returns the new array length.

---

```
unshift(elem1)
unshift(elem1, elem2)
...
unshift(elem1, elem2, ...)
```

---

#### Return value

Length of the calling array after the `unshift` call

#### Parameters

*elem1*, *elem2*, ...

Values to be inserted at the start of the calling array

This method modifies the calling object.



If the `unshift` method is applied to an object belonging to a class derived from the `Array` class, the inheritance principle is implicitly violated. This is because this method assigns the respective prototype value to the corresponding instance attribute for all array elements before execution.

#### Example

```
j=new Array();

k=j.unshift("a", "b", "c");
document.write(j + "<BR>");
document.write(k + "<BR>");

l=j.shift();
document.write(l + "<BR>");
document.write(j);
```

This example gives the following output:

```
["a","b","c"]
3
a
["b","c"]
```

### 7.1.16 valueOf method

This method returns a reference to the calling array object.

---

```
valueOf()
```

---

#### Return value

Reference to the calling array object

#### Parameters

None

#### *Example*

```
myArray = new Array("1","2","3");  
document.write(myArray.valueOf()[1]);
```

This example gives the following output:

2



## 7.2 Boolean class

An object of the `Boolean` class represents a logical value. There are no predefined attributes for this class.

### 7.2.1 Constructors

---

```
Boolean(expression)  
Boolean()
```

---

#### Return value

Object of the `Boolean` class

#### Parameters

*expression*

The expression *expression* is evaluated and converted to type `boolean` if necessary. An object with this value is created. If the constructor is called without an argument, an object with the value `false` is created.

### 7.2.2 equals method

This method compares the calling boolean object with the object transferred in the *object* argument for equality of class and value.

---

```
equals(object)
```

---

#### Return value

Boolean return value: `true` if the two objects are equal, `false` if not

#### Parameters

*object* Boolean object with which the calling boolean object is to be compared

### 7.2.3 `getClassName` method

This method returns a string specifying the name of the class to which the calling object belongs.

---

```
getClassName()
```

---

#### **Return value**

String specifying the class to which the calling object belongs, in this case `Boolean`

#### **Parameters**

None

#### *Example*

```
boolobj = new Boolean();  
boolClass = boolobj.getClassName(); // boolClass is a string containing  
                                     // the class name "Boolean"
```

### 7.2.4 `setValue` method

This method assigns a new value to the calling boolean object.

---

```
setValue(value)
```

---

#### **Return value**

None

#### **Parameters**

*value* New value for the calling boolean object

## 7.2.5 toString method

This method converts the boolean values `true` and `false` into the corresponding string values `"true"` and `"false"`.

---

```
toString()
```

---

### Return value

String `"true"` or `"false"`, depending on the value of the calling boolean object

### Parameters

None

### *Example*

```
boolVar = new Boolean(true);  
document.write(boolVar.toString());
```

This example gives the following output:

```
true
```

## 7.2.6 valueOf method

This method returns the boolean value `true` or `false`, depending on the value of the object.

---

```
valueOf()
```

---

### Return value

Boolean value `true` or `false`

### Parameters

None

### *Example*

```
boolVar = new Boolean(true);  
document.write(boolVar.valueOf());
```

This example gives the following output:

```
true
```

## 7.3 Date class

An object of the `Date` class represents a date.

### 7.3.1 Constructors

---

```
Date()  
Date(milliseconds)  
Date(year, month, day)  
Date(year, month, day, hour)  
Date(year, month, day, hour, minute)  
Date(year, month, day, hour, minute, second)
```

---

#### Return value

Object of the `Date` class

#### Parameters

If the constructor is called without an argument, the new object is initialized with the current date and time.

If a single argument (*milliseconds*) is specified, this is converted to type `number` and interpreted as the number of milliseconds since 1.1.1970 00:00:00 GMT. The new object is then initialized with this time. If the specified time lies outside the permitted range (1970-9999), the object is initialized with the current time.

For the arguments *month*, *day*, *hour*, *minute* and *second* the following ranges are valid:

```
month: 0-11  
day: 1-31  
hour: 0-23  
minute, second: 0-59
```

If three or more arguments are specified, these are converted to type `number` and interpreted as a local time specification in the format year, month, day, hour, minute, second. Missing arguments are set to 0 by default.

### 7.3.2 equals method

This method compares the calling date object with the object transferred in the *object* argument for equality of class, attributes, and values.

---

```
equals(object)
```

---

#### Return value

Boolean return value: `true` if the two objects are equal, `false` if not

#### Parameters

*object* Object with which the calling object is to be compared

### 7.3.3 getClassName method

This method returns a string specifying the name of the class to which the calling object belongs.

---

```
getClassName()
```

---

#### Return value

String specifying the class to which the calling object belongs, in this case `Date`

#### Parameters

None

### 7.3.4 get... methods

---

```
getDate()  
getDay()  
getHours()  
getMinutes()  
getMonth()  
getSeconds()  
getYear()
```

---

#### Return values

`getDate()` returns the day of the month.

`getDay()` returns the day of the week (in numerical form, i.e. Sunday = 0, Monday=1, etc.).

`getHours()` returns the hour.

`getMinutes()` returns the minute.

`getSeconds()` returns the second.

`getMonth()` returns the month (in numerical form, i.e. January = 0, February =1, etc).

`getYear()` returns the year

These methods return the value of the associated attribute in the local date and time.

#### Parameters

None

### 7.3.5getTimezoneOffset method

This method indicates the time difference between local time and GMT. The difference is given in minutes.

---

```
getTimezoneOffset()
```

---

#### Return value

number data type value in minutes giving the time difference.

#### Parameter

None

#### *Example*

```
christmasTime = new Date(2002, 11, 25);  
output = 'Local time on Christmas Day 2002 is different from GMT by ';  
output += christmasTime.getTimezoneOffset();  
output += ' minutes!';  
document.writeln(output);
```

In the Central European time belt the examples generates the following output:

```
Local time on Christmas Day 2002 is different from GMT by 60 minutes!
```



### 7.3.6 set... methods

---

```
setDate(Day)  
setHours(Hour)  
setMinutes(Minute)  
setMonth(Month)  
setSeconds(Second)  
setYear(Year)
```

---

These methods set the value of the associated attribute to the specified value.

#### Return value

None

#### Parameters

If the value specified for the argument *hour*, *minute*, *second*, or *month* is not valid, the argument is set to 0 by default. If the value specified for the argument *day* is not valid, this is set to 1. No check is carried out to establish whether the value specified in *day* is valid for the corresponding *month*. If the value specified in *year* lies outside the permitted range (1970-9999), the object is not modified.

All arguments must be specified in local time.

### 7.3.7 toGMTString method

This method returns a string in the format "Mon, Dec 20 1999 17:23:45 GMT" for the calling date object. In other words, it converts the local date and time to GMT.

---

```
toGMTString()
```

---

#### Return value

String containing the date of the calling object in the format described above

#### Parameters

None

### 7.3.8 toLocaleString method

This method returns a string in the format "12/20/99 17:23:45" containing the local date and time for the date object. Years prior to 2000 are shown using two figures, years after 2000 use all four figures.

---

```
toLocaleString()
```

---

#### Return value

String containing the date of the calling object in the format described above

#### Parameters

None

### 7.3.9 toString method

This method returns for the `Date` object a string in the format "Mon Dec 20 17:23:45 1999" in local time.

---

```
toString()
```

---

#### Return value

String containing the date of the calling object in the format described above

#### Parameters

None

#### Example

```
<wtoncreatescript>
<!--
d=new Date();
o=new Object();
o.d=d;
document.writeln('<br>',d.toString());
document.writeln('<br>',o.toString());
//-->
</wtoncreatescript>
```

The examples generates the following output:

```
Tue Jun 15 20:26:41 2010 {d: (new Date(1276626401171))}
```

### 7.3.10 valueOf method

This method returns a value of data type `number` for the date object. This value specifies the number of milliseconds since 01.01.1970 00:00:00h GMT.

---

```
valueOf()
```

---

#### Return value

Value of data type `number` which represents the value of the calling date object

#### Parameters

None

## 7.4 Document class

The `Document` class allows you access the HTML output stream and files.

Within `WTSript`, `Document` has no predefined attributes.

Within client-side JavaScript (which is executed in the browser and not on the `WebTransactions` system), you can of course use all attributes and methods of the document object which are defined in JavaScript.

### 7.4.1 Constructor

At least one instance of the `Document` class is always automatically available. This can be used to access the HTML output stream.

The constructor also allows you to generate instances of the `Document` class for accessing files.

---

`Document(filename)`

---

#### Return value

Instance of the `Document` class for accessing files

#### Parameters

*filename*

Name of the file you wish to access. This must be located under the base directory. Relative file names refer to the session directory. The system searches for files in the `Session` directory (`tmp/session-number`). You can also specify absolute path names.



The constructor creates an object of the `Document` class. The specified file is **not** opened implicitly (see [section “open method” on page 151](#)).

### 7.4.2 clear method

This method deletes the previous content of the output file or the HTML output stream. It has no return value.

It can be applied to the HTML output stream (`document.clear()`) and to objects that were created with `new Document(file)` and opened for writing with `open()` (i.e. without the `READ` parameter).

---

```
clear()
```

---

#### Return value

None

#### Parameters

None

### 7.4.3 close method

This method closes the output file currently open and returns a reference to the object. It can only be applied to objects created with `new Document(file)`, and not to the HTML output stream.

---

```
close()
```

---

#### Return value

Reference to the calling object

#### Parameters

None

### 7.4.4 equals method

This method compares the calling document object with the object transferred in the *object* argument for equality of class, attributes, and values.

---

```
equals(object)
```

---

#### Return value

Boolean return value: `true` if the two objects are equal, `false` if not

#### Parameters

*object* Object with which the calling object is to be compared

### 7.4.5 getClassName method

This method returns a string specifying the name of the class to which the calling object belongs.

---

```
getClassName()
```

---

#### Return value

String specifying the class to which the calling object belongs, in this case "Document"

#### Parameters

None

## 7.4.6 open method

This method opens the file specified in the constructor and returns a reference to the object. If it is not explicitly preceded by a `close()` call, the `close()` method is executed implicitly.

---

```
open()  
open(openmode)
```

```
openmode ::= "WRITE" [, "APPEND"] | "READ" | "R[EA]D[W[R]ITE]" [, "APPEND"] | "APPEND"
```

---

### Return value

Reference to the document object

### Parameters

`WRITE` (default)

The file is opened in write mode. You can then use the methods `write()` and `writeln()` to enter the output text in the file. If the file does not yet exist, it is created automatically.

`WRITE, APPEND`

The file is opened in write mode. You can then use the methods `write()` and `writeln()` to enter the output text in the file. If the file does not yet exist, it is created automatically. If the file already exists, the new text is appended to it.

`READ` The file is opened in read-only mode and must already exist.

`READWRITE`

The file must exist and is opened in write mode. You can then use the methods `write()` and `writeln()` to enter the output text in the file. The existing contents are deleted on the first `write[ln]()`.

`READWRITE, APPEND`

If the file does not yet exist, it is created automatically. If the file already exists, the new text is appended to it.

`APPEND`

The file is opened in write mode. You can then use the methods `write()` and `writeln()` to append the output text to the end of the file. If the file does not yet exist, it is created automatically. If the file already exists, the new text is appended to it.

### 7.4.7 read method

This method reads the complete contents of the file that is assigned to the document object and returns them. It can only be used for objects that were created with `new Document(file)` and opened for reading with `open()` and the parameter `READ` or `READWRITE`.

---

```
read()
```

---

#### Return value

Contents of the file assigned to the document object

#### Parameters

None

#### Example

```
file = new Document("../greetings.txt");
file.open("READ");
if (WT_SYSTEM.ERROR == "")
    str = file.read();
document.write(str);
```

### 7.4.8 valueOf method

This method returns the name of the file which is assigned to the document object. If the method is used with the predefined document object, then the string "`null`" is returned.

---

```
valueOf()
```

---

#### Return value

Name of the file assigned to the document object

#### Parameters

None



### 7.4.9 write / writeln method

These two methods output the arguments, converted to string form, in the output stream to the browser or to an open output file. This means that you can generate output within `OnCreate` and `OnReceive` scripts. `writeln` also generates a terminating line feed. However, since by default HTML simply treats line feeds as separators, `writeln` only generates line feeds in the case of preformatted text, e.g. within `<PRE>` tags.

---

```
write()
write(expression)
write(expression, expression)
...
write(expression, expression, ...)
```

---

```
writeln()
writeln(expression)
writeln(expression, expression)
...
writeln(expression, expression, ...)
```

---

#### Return value

None

#### Parameters

*expression*

One or more expressions to be written to the output stream in string format. `write` and `writeln` can be called with any number of arguments.

#### Example

```
document.write("Good");
document.write("Morning");
document.write("<BR>");
document.writeln("Good");
document.writeln("Morning");
```

This example gives the following output:

(Output as part of preformatted text:)

```
GoodMorning
Good
Morning
```

(Output outside preformatted text:)

```
GoodMorning
Good Morning
```

## 7.5 Host data object class

All the data which WebTransactions receives from or sends to the host application is stored in host data objects (for more information, see the WebTransactions manual “Concepts and Functions”).

When messages are received from the host, host data objects are created as attributes of the corresponding communication object. There are no explicit constructors.

### 7.5.1 `getClassName` method

This method returns a string specifying the name of the class to which the calling object belongs.

---

```
getClassName()
```

---

#### **Return value**

String specifying the class to which the calling object belongs, in this case "WT\_OldHostobject".

#### **Parameters**

None

## 7.5.2 toString method

This method evaluates the calling object in accordance with the class template and returns the result as a string.

---

```
toString()  
toString(expression)
```

---

### Return value

String containing the result of the class template evaluation

### Parameters

*expression*

The class template *expression*.`clt` is used. This allows you to define additional class templates independently of the type of host data object and call these as required.

If no argument is specified, WebTransactions uses the class template which corresponds to the type of the calling host data object (*type*.`clt`). The possible types depend on the communication module: in the case of openUTM, the `IOTYPE` attribute is evaluated whereas for OSD and MVS, the `Type` attribute is also evaluated.



For further information on class templates, see [chapter “Class templates \(\\*.clt\)” on page 309](#).

The evaluation operator is described in [section “Evaluation operator ##...#” on page 79](#).

### 7.5.3 valueOf method

This method returns a reference to the object itself.

---

```
valueOf()
```

---

#### **Return value**

Reference to the calling host object

#### **Parameters**

None

## 7.6 Function class

The `Function` class is used to define and handle functions as objects. It recognizes a constructor and attributes, but not methods for `Function` objects.

Dynamically generated functions have two advantages over interpreted functions:

- They run quicker as quite a few operations need only be performed once.
- The function body can be generated dynamically, i.e. depending on the data available.

### 7.6.1 Constructors

---

```
Function(body)  
Function(arg1, body)  
Function(arg1, arg2, body)  
...  
Function(arg1, ..., argn, body)
```

---

#### Return value

Instance of the `Function` class

#### Parameters

*arg1*, ..., *argn*

Formal arguments of the new function

*body* String with WTSript statements

The constructor creates a new object of the `Function` class with the optional formal arguments *arg1* through *argn* and the statements defined in *body*.

*Example*

The following functions are to be created using the constructor, rather than by means of a fixed definition:

```
function f1(no) {
    document.writeln( no.getDay()+' '.substring(0,4),
                      no.getMonth()+' '.substring(0,4),
                      no.getYear()+' '.substring(0,6),
                      no );
}
```

A corresponding function created using the constructor might look like this, for example:

```
f1 = new Function("no", "{document.writeln ( no.getDay()+' '.substring(0,4)
+ no.getMonth()+' '.substring(0,4)
+ no.getYear()+' '.substring(0,6)
, no );}");
```

## 7.6.2 Attributes

---

```
arity
caller
prototype
arguments
callee
```

---

```
arity
```

This attribute contains the number of defined arguments.

```
caller
```

The local variable `caller` contains a reference to the calling function. It is valid only when it appears in the function body. If the function was called from the top script level, `caller` has the value `null`.

```
prototype
```

This attribute contains all class properties, which are inherited by all instances of that class.

```
arguments
```

When processing the constructor, the `arguments` attribute provides an array which can be used to access the current parameters of the function.

```
callee
```

This attribute contains a reference to the function itself. It is valid only when it appears in the function body, and is used to create recursive calls.

*Example*

The following example demonstrates how to use the `Function` class to define a recursive function:

```

functStr = "{if (a == 3) { b += a;}
            calleeTyp = typeof callee;
            functCallee = callee;
            functCalleeString = callee.toString();
            if (caller)
            { callerTyp = typeof caller;
              functCaller = caller;
              functCallerString = caller.toString();
            }
            if (a == 0 || b == 0)
            { document.writeln("<TR><TD>Function terminated prematurely as
              argument = 0!</TD></TR>");
              return 0;
            }
            document.writeln("<TR><TD>Recursive call of callee!</TD><TD>\"
              + callee(0, 0) + \"</TD></TR>");
            return b;
          }";
functPar1 = "a";
functPar2 = "b";
funct01 = new Function (functPar1, functPar2, functStr);
document.writeln("<TABLE>");
document.writeln("<TR><TD>Number of function parameters for
  funct01():</TD><TD>" + funct01.arity + "</TD></TR>");
document.writeln("<TR><TD>Names of the function parameters:</TD></TR>");
document.writeln("<TR><TD>1. ," + functPar1.toString() + "';</TD><TD>2. ," +
  functPar2.toString() + "';</TD></TR>");
document.writeln("<TR><TD> Here are the <BIG>calls</BIG> of
  funct01()!</TD></TR>");
document.writeln("<TR><TD> funct01(4, 2) returns:</TD><TD>" + funct01(4, 2) +
  "</TD></TR>");
document.writeln("<TR><TD> funct01(3, 2) returns:</TD><TD>" + funct01(3, 2) +
  "</TD></TR>");

```

This example gives the following output:

```

Number of function parameters for funct01():    2
Names of the function parameters:
1. ,a';    2. ,b';
Here are the calls of funct01()!
Function terminated prematurely as argument = 0!
Recursive call of callee!    0
funct01(4, 2) returns:    2
Function terminated prematurely as argument = 0!

```

```
Recursive call of callee!    0
funct01(3, 2) returns:      5
```

### 7.6.3 equals method

This method compares the calling function object with the object transferred in the *object* argument for equality of class, attributes, and values.

---

```
equals(object)
```

---

#### Return value

Boolean return value: `true` if the two objects are equal, `false` if not

#### Parameters

*object* Object with which the calling object is to be compared

### 7.6.4 getClassName method

This method returns a string specifying the name of the class to which the calling object belongs.

---

```
getClassName()
```

---

#### Return value

String specifying the class to which the calling object belongs, in this case `Function`

#### Parameters

None



## 7.7 Math class

The `Math` class is used for mathematical operations and does not permit instantiation. It therefore has no constructor. The methods are available only as class methods, i.e. they are called directly using the class name: `Math.function()`.

### 7.7.1 Class attributes

The following class attributes exist.

---

<code>E</code>	Base of the natural logarithm (Eulerian number $e$ , approx. 2.718281828)
<code>LN10</code>	Natural logarithm of 10 (approx. 2.302585)
<code>LN2</code>	Natural logarithm of 2 (approx. 0.693147)
<code>LOG2E</code>	Logarithm to the base 2 of $E$ (approx. 1.442695)
<code>LOG10E</code>	Decimal logarithm of $E$ (approx. 0.434294)
<code>PI</code>	Value of $\pi$ (approx. 3.141592)
<code>SQRT2</code>	Square root of 2 (approx. 1.414213)
<code>SQRT1_2</code>	Square root of $\frac{1}{2}$ (approx. 0.707106)

---

### 7.7.2 `abs` method

This method returns the absolute value of *number*.

---

`abs(number)`

---

#### Return value

Absolute value of *number*

#### Parameters

*number* Any floating-point number whose absolute value is to be returned

### 7.7.3 **acos method**

This method returns the arccosine of the value specified in *number*.

---

```
acos(number)
```

---

#### **Return value**

If the argument is a value between -1 and +1, the result is returned in the radian 0 to  $+\pi$ .

Otherwise, NaN is returned.

#### **Parameters**

*number* Floating-point number between -1 and +1

### 7.7.4 **asin method**

This method returns the arcsine of the value specified in *number*.

---

```
asin(number)
```

---

#### **Return value**

If the argument is a value between -1 and +1, the result is returned in the radian  $-\pi/2$  to  $+\pi/2$ .

Otherwise, NaN is returned.

#### **Parameters**

*number* Floating-point number between -1 and +1

### 7.7.5 atan method

This method returns the arctangent of the value specified in *number*.

---

`atan(number)`

---

#### Return value

The result is returned in the radian  $-\pi/2$  to  $+\pi/2$ .

#### Parameters

*number* Floating-point number between  $-\text{Infinity}$  and  $+\text{Infinity}$

### 7.7.6 ceil method

This method returns the lowest integer that is not less than the value specified in *number*.

---

`ceil(number)`

---

#### Return value

If *number* is an integer, the result is identical to the argument. Otherwise, the next highest integer to *number* is returned.

#### Parameters

*number* Any floating-point number

#### Example

```
document.writeln("<BR>ceil(-3.14) is " + Math.ceil(-3.14) + "!");  
document.writeln("<BR>ceil(3.14) is " + Math.ceil(3.14) + "!");
```

This example gives the following output:

```
ceil(-3.14) is -3!  
ceil(3.14) is 4!
```

### 7.7.7 cos method

This method returns the cosine of the value specified in *number*.

---

`cos(number)`

---

#### Return value

Value between -1 and 1

#### Parameters

*number* Floating-point number in the radian

### 7.7.8 exp method

This method returns  $e^{number}$  (e to the power of *number*)

---

`exp(number)`

---

#### Return value

$e^{number}$  (e to the power of *number*)

#### Parameters

*number* Floating point number

### 7.7.9 floor method

This method returns the highest integer that is not greater than the value specified in *number*.

---

`floor(number)`

---

#### Return value

If *number* is an integer, the result is identical to the argument. Otherwise, the next lowest integer to *number* is returned.

#### Parameters

*number* Floating point number

#### Example

```
document.writeln("<BR>floor(-3.14) is " + Math.floor(-3.14) + "!");  
document.writeln("<BR>floor(3.14) is " + Math.floor(3.14) + "!");
```

This example gives the following output:

```
floor(-3.14) is -4!  
floor(3.14) is 3!
```

### 7.7.10 log method

This method returns the natural logarithm of the value specified in *number*.

---

`log(number)`

---

#### Return value

Natural logarithm of the value specified in *number*. If *number* is less than 0, NaN is returned.

#### Parameter

*number* Floating point number whose natural logarithm is to be returned

### 7.7.11 max method

This method returns the greater of two floating-point numbers.

---

```
max(number1, number2)
```

---

#### Return value

Value of the greater of the two arguments

#### Parameters

*number1*

*number2*

Floating-point numbers whose maximum is to be returned

### 7.7.12 min method

This method returns the lesser of two floating-point numbers.

---

```
min(number1, number2)
```

---

#### Return value

Value of the lesser of the two arguments

#### Parameters

*number1*

*number2*

Floating-point numbers whose minimum is to be returned

### 7.7.13 pow method

This method returns a floating-point number raised to the power of another floating-point number.

---

```
pow(number1, number2)
```

---

#### Return value

$number1^{number2}$  (*number1* to the power of *number2*), provided the arguments are valid. Otherwise, NaN is returned.

#### Parameters

*number1*

Any floating-point number

*number2*

Any floating-point number

### 7.7.14 random method

This method returns a pseudo random number.

---

```
random()
```

---

#### Return value

A random number of type `number`, which is derived from the current time.

This value will be greater than or equal to 0 and less than 1.

#### Parameters

None

### 7.7.15 round method

This method returns the value specified in *number* rounded to the nearest integer. The value .5 is rounded up to the nearest integer.

---

`round(number)`

---

#### Return value

Value specified in *number* rounded off to the nearest integer

#### Parameters

*number* Any floating-point number

#### Example

```
document.writeln("<BR>round(0.500) is " + Math.round(0.5) + "!");  
document.writeln("<BR>round(0.499) is " + Math.round(0.499) + "!");  
document.writeln("<BR>round(-0.500) is " + Math.round(-0.5) + "!");  
document.writeln("<BR>round(-0.501) is " + Math.round(-0.501) + "!");
```

This example gives the following output:

```
round(0.500) is 1!  
round(0.499) is 0!  
round(-0.500) is 0!  
round(-0.501) is -1!
```



### 7.7.16 **sin method**

This method returns the sine of a floating-point number.

---

```
sin(number)
```

---

#### **Return value**

Sine of the value specified in *number*. The result is a value between -1 and 1.

#### **Parameters**

*number* Any floating-point number

### 7.7.17 **sqrt method**

This method returns the square root of the value specified in *number*.

---

```
sqrt(number)
```

---

#### **Return value**

Square root of the value specified in *number*, provided *number*  $\geq 0$ . Otherwise, NaN is returned.

#### **Parameters**

*number* Floating-point number  $\geq 0$

### 7.7.18 **tan method**

This method returns the tangent of a floating-point number.

---

```
tan(number)
```

---

#### **Return value**

Tangent of the value specified in *number*

#### **Parameters**

*number* Any floating-point number

## 7.8 Number class

An object belonging to the `Number` class represents a numerical value.

### 7.8.1 Constructors

---

```
Number(expression)  
Number()
```

---

#### Return value

Object of type `Number`

#### Parameters

*expression*

The expression *expression* is evaluated and converted to type `number` if necessary. An object with the corresponding value is created.

If *expression* is not specified then an object with the value `0` is created.

### 7.8.2 Class attributes

---

<code>MAX_VALUE</code>	The largest value that can be represented in <code>WebTransactions</code>
<code>MIN_VALUE</code>	The smallest positive amount that can be represented in <code>WebTransactions</code>
<code>NaN</code>	The value <code>NaN</code> (not a number)

---

These attributes are class attributes. Objects belonging to the `Number` class do not possess these attributes.

#### Example

```
my_number=new Number();  
document.write(my_number.NaN); //Output: undefined  
document.write(Number.NaN);   //Output: NaN
```

### 7.8.3 equals method

This method compares the calling number object with the object transferred in the *object* argument for equality of class and value.

---

```
equals(object)
```

---

#### Return value

Boolean return value: `true` if the two objects are equal, `false` if not

#### Parameters

*object* Object with which the calling object is to be compared

### 7.8.4 getClassName method

This method returns a string specifying the name of the class to which the calling object belongs.

---

```
getClassName()
```

---

#### Return value

String specifying the class to which the calling object belongs, in this case `Number`

#### Parameters

None

#### Example

```
myNumber = new Number();  
document.write(myNumber.getClassName());
```

This example gives the following output:

Number

### 7.8.5 setValue method

This method assigns a new value to the calling number object.

---

```
setValue(value)
```

---

#### Return value

None

#### Parameters

*value* New value for the calling number object

### 7.8.6 toString method

This method returns the numerical value of the calling object converted back into string format.

---

```
toString()
```

---

#### Return value

String representing the value of the calling object

#### Parameters

None

### 7.8.7 valueOf method

This method returns the value of the calling object in `number` format

---

```
valueOf()
```

---

#### Return value

Value of type `number` which represents the value of the calling number object

#### Parameters

None

## 7.9 Object class

Objects of the `Object` class are containers for known attributes. You can therefore create as many attributes as necessary. However, there are no predefined attributes.

### 7.9.1 Constructors

---

```
Object()  
Object(expression)
```

---

If you call the constructor without specifying *expression*, it creates an object of the class `Object`. This is an empty container for which attributes can now be created.

If the constructor is called with *expression* specified, then *expression* is evaluated. If the expression is a reference to an object, then a new reference to this object is created. Otherwise a new object belonging to the class `Boolean`, `Number`, or `String` is created depending on the type of the expression.

#### *Example*

```
a = new Object();           //creates a new object of class Object  
b = new Object(false);     //creates a new object of class Boolean  
c = new Object(a);         //supplies a reference to object a,  
                           //not a new object!
```



Objects of the `Object` class can also be created directly:

```
d = {attr:wert, attr:wert};
```

### 7.9.2 equals method

This method compares the calling object with the object transferred in the *object* argument for equality of class, attributes, and values.

---

```
equals(object)
```

---

#### Return value

Boolean return value: `true` if the two objects are equal, `false` if not

#### Parameters

*object* Object with which the calling object is to be compared

#### Example

```
obj01 = new Object();
obj01.num = 42;
obj01.str = "forty two";
obj01.bool = true;
obj02 = new Object();
obj02.num = 21+21;
obj02.str = "forty two";
obj02.bool = true;
if ( obj01.equals( obj02 ) ) // returns true
. . .
```

### 7.9.3 getClassName method

This method returns a string specifying the name of the class to which the calling object belongs.

---

```
getClassName()
```

---

#### Return value

String specifying the class to which the calling object belongs, in this case `Object`

#### Parameters

None

## 7.9.4 toString method

This method returns a string listing each attribute and the associated value. If an attribute is an object of type `function`, then the function definition is returned instead of the value.

You can use this method to create a new object with identical attribute values.

In order to avoid endless chains, the `toString` method will terminate output in the event of recursion, i.e. output will be stopped as soon as the same object reference is encountered a second time.

---

`toString()`

---

### Return value

String listing each attribute and the associated value

### Parameters

None

*Example 1: Output with recursion*

```
a = new Object();
b = new Object();
a.str = "String a";
a.b = b;
b.str = "String b";
b.a = a;

document.write( a.toString() );
// Output: {b: {a: {}}, str: "String b"}, str: "String a"}
```

*Example 2*

```

<wtoncreatescript>
<!--
document.write("<br><br><h3>*****Object Test*****</h3><br>");

obj = new Object();
obj.b = "hello \"World!\\"";
obj.c = new String ("hello World!");
obj.d = new String ("hello \"World!\");
obj.e = new Number(21+21);
obj.f = new Boolean(true);
obj.g = false;
obj.o = new Object();
obj.o.text="Hallo";
obj.h = new Function("param", "{if (wt_system.abc)
wt_system.abc+=42; else wt_system.abc=0;}");
obj.i = new Document("file");
obj.j = 84/2;
obj.k = new Date(2010,1,1);
obj.l = this_variable_does_not_exist;
obj.m = null;
objString = obj.toString();
document.writeln(objString);

//Save object
file = new Document("../..objsave.txt");
file.open("WRITE");
file.write(objString);
file.close();

//Restore object
file = new Document("../..objsave.txt");
file.open("READ");
newStr= file.read();
eval("neuesObj = " + newStr + "");

//It is now possible to access the new object in the same way as the old one
document.write("<br><h3>*****Zugriff auf neues Objekt*****</h3>");
document.write(neuesObj.o.text);
//-->
</wtoncreatescript>
<wtoncreatescript>
<!--

// This is an example for functions only
obj1 = new Object();
obj1.f = new Function("param", "{if (wt_system.abc) \
wt_system.abc+=42; else wt_system.abc=0;}");
objString = obj1.toString();

```



```
document.write("<br><h3>*****Ausgabe Funktions-Objekt*****</h3>");
document.writeln(objString);
wt_system.abc=42;
neuesObj.h();
document.write("<br><h3>*****Ausgabe Aufruf Funktionsaufruf*****</h3>");
document.write(wt_system.abc);
//-->
</wtoncreatescript>
```

This example gives the following output:

#### \*\*\*\*\*Object Test\*\*\*\*\*

```
{b: "hello \"World!\"", c: (new String("hello World!")), d: (new
String("hello \"World\"!")), e: (new Number(42)), f: (new Boolean(true)), g:
false, h: function (param){if (wt_system.abc)
wt_system.abc+=42; else wt_system.abc=0;}, i: (new Document("file")), j: 42,
k: (new Date(1264978800000)), l: undefined, m: null, o: {text: "Hello"}}
```

#### \*\*\*\*\*Access to new object\*\*\*\*\*

Hello

#### \*\*\*\*\*Output of function object\*\*\*\*\*

```
{f: function (param){if (wt_system.abc) wt_system.abc+=42; else
wt_system.abc=0;}}
```

#### \*\*\*\*\*Output of function call\*\*\*\*\*

84

*Example 3*

```
o=new Object();
  o.s='abc'; o.n=42; o.b=true;
  o.S=new String('ABC'); o.N=new Number(43); o.B=new Boolean(false);
  o.D=new Date();
  o.u=a;           // a ist nicht definiert
  o.nu=null;
  o.O={n:1, m:2};
  o.A=[1,2,,3,,];
  document.writeln(o.toString());
```

**The example generates the following output:**

```
{A: [1,2,,3,,], B: (new Boolean(false)), D: (new Date(1275668431340)), N:
(new Number(43)), O: {m: 2, n: 1}, S: (new String("ABC")), b: true, n: 42,
nu: null, s: "abc", u: undefined}
```

### 7.9.5 valueOf method

This method returns a reference to the object itself.

---

```
valueOf()
```

---

#### **Return value**

Reference to the calling object itself

#### **Parameters**

None

## 7.10 RegExp class

An object of the `RegExp` class contains the specification of a regular expression. Regular expressions are search patterns which are used to locate specific character combinations in strings.

### 7.10.1 Constructors

---

```
RegExp()  
RegExp(expression)  
RegExp(expression, mode)  
variable = RegExpLiteral
```

---

#### Return value

Object of type `RegExp`

*expression*

Expression of type `string` which specifies the regular expression (see [section “Literals for regular expressions” on page 38](#)).

*mode* Expression of type `string` which specifies the mode:

i (ignore) Ignore uppercase/lowercase.

g (global) Find all suitable matches.

ig or gi

Ignore uppercase/lowercase and find all suitable matches.

*RegExpLiteral*

Assigning a literal for a regular expression also creates an object of class `RegExp`.

Note that while a string is specified in the `RegExp` constructor function, a regular expression is specified when you make a direct assignment using *RegExpLiteral* (see examples on the following pages). String literals must be enclosed in quotes, whereas `RegExp` literals must be delimited by slashes.

Except in escape sequences, backslashes are of no significance in string literals, i.e.

"\char" means the same as "char" and "a\*" means the same as "a\\*" (see [page 35](#)).

In contrast, a backslash in a `RegExp` literal invalidates the following metacharacter. In this way, for example, you can use `/a\*/` to search for the character sequence "a\*" but not for a sequence of a's. If you want to define this type of search pattern using the constructor function, you must specify the string "a\\*" as an argument.

If you want to use two-character metacharacters in string literals (e.g. `\d` as a metacharacter representing any digit), you must enter a double backslash. This is because `\char` and `char` have the same meaning in string literals. `/\d/` therefore has the same meaning as `new RegExp("\\d")`.

### *Examples*

In each case, the assignment of the literal generates the same regular expression as the constructor function call which follows it:

```
re_1 = /abc/i;  
re_1 = new RegExp("abc","i");  
  
re_2 = /ab+c/;  
re_2 = new RegExp("ab+c");  
  
re_3 = /\w+/  
re_3 = new RegExp("\\w+");  
  
re_4 = /\d/  
re_4 = new RegExp("\\d");
```

## 7.10.2 Attributes of objects of the RegExp class

Every object in the `RegExp` class possesses the following attributes after creation:

Attribute	Data type	Meaning
<code>source</code>	string	Specifies the search pattern which was entered as the source without surrounding slashes and without flags (i or g).
<code>ignoreCase</code>	boolean	Specifies whether the i flag for ignoring uppercase/lowercase is set (true) or not (false).
<code>global</code>	boolean	Specifies whether the g flag for global search is set (true) or not (false).
<code>lastIndex</code>	number	Index of the character at which the next search begins. <code>lastIndex</code> is only set for global searches. Before the first search, <code>lastIndex=0</code> . If another search is to be performed and if the new search is once again to start at the beginning of the string, you must first reset <code>lastIndex</code> to 0. The rules listed below apply to <code>lastIndex</code> .



You cannot modify the `source` and `ignoreCase` attributes by means of a direct assignment. To do this, you must use the `compile` method (see [section “compile method” on page 184](#)).

### Rules for `lastIndex`

The following rules apply to `lastIndex`:

- If `lastIndex` is greater than the length of the string to be searched through, the method `regexp.test` returns the value `false` and `regexp.exec` the value `null`. `lastIndex` is set to 0.
- If `lastIndex` is equal to the length of the string to be searched and the regular expression corresponds to an empty string, the method `regexp.test` returns the value `true` and `regexp.exec` returns a result array. `lastIndex` is unchanged.
- If `lastIndex` is equal to the length of the string to be searched and the regular expression does **not** correspond to an empty string, the method `regexp.test` returns the value `false` and `regexp.exec` the value `null`. `lastIndex` is set to 0.
- In all other cases, `lastIndex` is set to the index position which follows the last located match.

### 7.10.3 Predefined RegExp object

A predefined `RegExp` object is available in all templates. `WebTransactions` stores partial results in the attributes of this object after the following method calls:

- `RegExp` method calls `exec` and `test`
- `String` method calls `match` and `replace`

Although this predefined object has the name `RegExp`, it belongs to the `Object` class. It has the following attributes (all of type `string`):

Attribute	Meaning
<code>lastMatch</code>	Character sequence of last located match (= position)
<code>leftContext</code>	Substring to the left of the position
<code>rightContext</code>	Substring to the right of the position
<code>\$1,\$2..\$9</code>	<p>The character sequence at the located position which corresponds to the <math>n</math>th bracketed partial expression of the search pattern is stored in <math>\\$n</math> (see <a href="#">“Example 2” on page 202</a>) - provided that the search pattern does not contain more than nine bracketed partial expressions.</p> <p>Although there is no limit to the number of bracketed partial expressions which a search pattern can contain, the predefined <code>RegExp</code> object can only “remember” nine of them. If the search pattern contains more than nine bracketed partial expressions, then the matches for the last nine bracketed partial expressions are stored in <code>\$1-\$9</code>. You can access the matches for all the bracketed partial expressions via the indices of the associated result array.</p>
<code>lastParen</code>	The character sequence at the located position which corresponds to the last bracketed partial expression of the search pattern is stored in <code>lastParen</code> .
<code>input</code>	Default string which is used if no string to be searched through is specified when the <code>RegExp</code> methods <code>exec</code> or <code>test</code> are called. If you want to use the <code>input</code> attribute, you must assign it a string expression which specifies the default search pattern.

### 7.10.4 compile method

This method compiles a search pattern which is specified by a string expression. The arguments correspond to those of the constructor function.

---

```
compile(expression)  
compile(expression, mode)
```

---

#### Return value

Reference to the calling object itself

#### Parameters

*expression*

Expression of type `string` which specifies the regular expression (see [section “Literals for regular expressions” on page 38](#))

*mode* Expression of type `string` which specifies the mode:

`i` (`ignoreCase`) Ignore uppercase/lowercase.

`g` (`global`) Find all suitable matches.

`ig` or `gi`

Ignore uppercase/lowercase and find all suitable matches.

You can use the `compile` method to assign a new regular expression to an existing `RegExp` object. The arguments correspond to those used in the constructors. The following object attributes are reset: `source`, `ignore`, `global`, and `lastIndex`. The regular expression is stored in compiled form in the calling object.



### 7.10.5 equals method

This method compares the calling object with the object transferred in the *object* argument for equality of class, attributes, and values.

---

```
equals(object)
```

---

#### Return value

Boolean return value: `true` if the two objects are equal, `false` if not

#### Parameters

*object* Object with which the calling object is to be compared

### 7.10.6 exec method

This method searches in the specified string in accordance with the calling regular expression.

Each `exec` method call returns a maximum of one match:

- If the `global` flag of the regular expression is not set (i.e. `regexp.global=false`), then the `exec` method always returns the first match. In this case, the call `regexp.exec(searchStr)` corresponds to the call `searchStr.match(regExp)`.
- If the `global` flag of the regular expression is set (`=true`) and if `exec` is called a number of times in succession, then the next match is found on each call. The search always starts after the last located match. The left-hand context also starts after the preceding located match and not at the start of the string.

If the `i` flag of the regular expression is set (`=true`), then the uppercase/lowercase distinction is ignored for the purposes of the comparison.

---

```
regexp.exec()  
regexp.exec(string)
```

---

#### Return value

If a match is found, `exec` returns a result array and sets the attributes of the predefined `RegExp` object to the current values. If the `g` flag is set (i.e. `regexp.global=true`), then the `lastIndex` attribute of the calling object `regexp` is reset.

If no match is found, the value `null` is returned and the attribute values of the predefined `RegExp` object are deleted.

#### Parameters

*regexp* Regular expression. This can be the name of an object or a literal.

*string* (optional) String expression to be searched through. If *string* is not already of type `string` then it is converted to this type. If *string* is not specified, `RegExp.input` is used.

## Result array of the exec method

The `exec` method returns an array with the following attributes:

<code>[0]</code>	Substring of <i>string</i> matching the regular expression
<code>[n]</code>	Concrete contents of <i>n</i> =1st 2nd ... <i>n</i> th brackets of the regular expression
<code>index</code>	Index of the first character of the located match
<code>input</code>	String <i>string</i>

*Example: exec call first without and then with g flag*

The following regular expression searches for double characters.

```
regTest = /((\w)\2)/i;
```

The outside brackets are not essential but make the pattern easier to follow. The word "Dampfschiffahrtsgesellschaft" is to be searched through:

```
text = "Dampfschiffahrtsgesellschaft";
result = regTest.exec(text);
```

This gives the following result:

- Attributes of the `RegExp` object `regTest`:

```
global: false
ignoreCase: true
lastIndex: 11
source: "((\w)\2)"
```

- Attributes of the result array `result`:

```
index: 9
input: "Dampfschiffahrtsgesellschaft"
0: "ff"
1: "ff"
2: "f"
```

- Attributes of the predefined `RegExp` object:

```
$1: "ff"
$2: "f"
lastMatch: "ff"
lastParen: "f"
leftContext: "Dampfschi"
rightContext: "fahrtsgesellschaft"
```

If we call `regTest.exec(text)` again, we obtain the same results since the `g` flag is not set.

In contrast, if we define a **global search**:

```
regTest = /((\w)\2)/ig;
```

then only the first `exec` call returns the results presented above. After the second call we get:

– **Attributes of the RegExp object** `regTest`:

```
global: true
ignoreCase: true
lastIndex: 23
source: "((\w)\2)"
```

– **Attributes of the result array** `result`:

```
index: 21
input: "Dampfschiffahrtsgesellschaft"
0: "11"
1: "11"
2: "1"
```

– **Attributes of the predefined object** `RegExp`:

```
$1: "11"
$2: "1"
lastMatch: "11"
lastParen: "1"
leftContext: "fahrtsgese"
rightContext: "schaft"
```

and on the third call:

– **Attributes of the RegExp object** `regTest`:

```
global: true
ignoreCase: true
lastIndex: 0
source: "((\w)\2)"
```

– `result = null`

– **Attributes of the predefined RegExp object**:

```
lastMatch: ""
lastParen: ""
leftContext: ""
rightContext: ""
```

### 7.10.7 **getClassName** method

This method returns a string specifying the name of the class to which the calling object belongs.

---

```
getClassName()
```

---

#### **Return value**

String specifying the class to which the calling object belongs, in this case "RegExp".

#### **Parameters**

None

### 7.10.8 test method

This method checks whether the string *string* contains a substring which matches the regular expression. If it does, `true` is returned; otherwise `false`. If the `g` (global) flag is set in the calling object, the search starts at `lastIndex`. Otherwise, it starts at the beginning of *string*. If no argument is specified, `RegExp.input` is used.

---

```
test(string)  
test()
```

---

#### Return value

`true` if the string specified in *string* contains a substring which matches the regular expression, or `false` otherwise

#### Parameters

*string*

String to be searched to establish whether or not it contains a substring that matches the regular expression. This argument is converted to type `string`.

#### Example

```
function check(str,re) {  
  if(re.test(str))  
    result=" contains the pattern: ";  
  else  
    result=" does not contain the pattern: ";  
  document.write(str+result+re.source);  
}
```

```
check("WebTransactions",/bt/i);
```

The call checks whether `WebTransactions` contains a match for the regular expression `/bt/i`. Since the `i` flag is set, uppercase/lowercase notation is ignored and a match is found. The output also indicates that there are no flags present in the `re.source` property:

```
WebTransactions contains the pattern: bt
```

## 7.11 String class

A string object represents a string.

A string can be considered to be a type of character array: you can address each individual character via its index.

The string constructor makes it possible to initiate the conversion of an expression into a string. You can only create additional individual attributes for a string object which has been created by means of a constructor.

### 7.11.1 Constructors

---

```
String(expression)  
String()
```

---

#### Return value

Object of the `String` class

The expression *expression* is evaluated and converted to type `string`. An object with the corresponding value is created. If *expression* is not specified then an object with the value empty string is created.

### 7.11.2 Attributes

---

```
length  
[number]
```

---

`length` specifies the current length of the string.

You can use the index expression [*number*] to address the individual characters in the string. Thus, for example, `a[7]` returns the 8th character of the string object `a`.

### 7.11.3 charAt method

This method returns the character located at the position in the calling string indicated by *Index*.

---

```
charAt(Index)
```

---

#### Return value

The result is returned as a string. If the value of *Index* does not lie between 0 and *callingString*.length - 1, then an empty string is returned.

#### Parameters

*Index* Expression which specifies the position in the calling string object at which the character to be returned is located

#### Example

```
document.write("WebTransactions".charAt(0)); // 1st character
document.write("WebTransactions".charAt(16)); // Index greater than
// string length
document.write("WebTransactions".charAt(4-1)); // 4th character
```

This example gives the following output:

WT



### 7.11.4 charCodeAt method

This method returns the numeric code of the character located at the position specified by the expression *n* in the calling String object.

---

```
charCodeAt(n)
```

---

#### Return value

The result is returned as a number or numeric value. If *n* does not lie between 0 and *callingString*.length - 1, NaN (not a number) is returned.

#### Parameter

*n* Expression specifying the position of the character to be returned in the calling string object.

#### Example

```
NL = String.fromCharCode(015,012); // octal
document.writeln("<br>NL[0]=" ,NL.charCodeAt(0), ' ,
', "NL[1]=" ,NL.charCodeAt(1) );
```

This example generates the following output:

```
NL[0]=13, NL[1]=10
```

### 7.11.5 concat method

This method joins the calling string with the string specified in *string*

---

```
concat(string)
```

---

#### Return value

The concatenated string. Neither the calling string object nor the string transferred in the *string* argument is modified.

#### Parameters

*string* String with which the calling string object is to be joined

#### Example

```
strObj = new String ("Good Morning");  
str = "Goodbye";  
resStr1 = strObj.concat(", Mr. President!");  
resStr2 = str.concat(", Mr. President!");  
document.write("<BR>" + resStr1);  
document.write("<BR>" + resStr2);
```

This example gives the following output:

```
Good Morning, Mr. President!  
Goodbye, Mr. President!
```

### 7.11.6 equals method

This method compares the calling string object with the object transferred in the *object* argument for equality of class and value.

---

```
equals(object)
```

---

#### Return value

Boolean return value: `true` if the two objects are equal, `false` if not

#### Parameters

*object* Object with which the calling object is to be compared

### 7.11.7 fromCharCode method

This method returns a string formed from the codes of the numeric codes passed as arguments. `fromCharCode()` is a static method of the `String` constructor.

---

```
String.fromCharCode(code1 [, code2, [...]] );
```

---

#### Return value

The result is returned as a string.

#### Parameters

*code1*, *code2*, ...

Numeric codes from which the output string is to be formed.

#### Example

```
NL = String.fromCharCode(13,10); //decimal
NL = String.fromCharCode(0x0d,0x0a); //hex
NL = String.fromCharCode(015,012); //octal
WT_SYSTEM.HTTP_HEADER = 'Content-type: text/html'+NL+'Connection:
close'+NL+NL;
```

This example generates the following output:

```
HTTP_HEADER: Content-type: text/html
              Connection: close
```

### 7.11.8 `getClassName` method

This method returns a string specifying the name of the class to which the calling object belongs.

---

```
getClassName()
```

---

#### **Return value**

String specifying the class to which the calling object belongs, in this case `String`

#### **Parameters**

None

### 7.11.9 indexOf method

This method searches the calling string object from beginning to end for the string specified in *searchValue*. If *fromIndex* is specified, then the search starts at this index. Otherwise, it starts at 0. While searching, the method distinguishes between uppercase and lowercase.

---

```
indexOf(searchValue)  
indexOf(searchValue, fromIndex)
```

---

#### Return value

Index of the first character of the match closest to the start of the string, or -1 if *searchValue* is not found

#### Parameters

*searchValue*

String to be searched for

*fromIndex*

Position in the calling string object at which the search for the string specified in *searchValue* is to begin

Default: 0

#### Example

```
str = "The a the b and the c."  
document.write(str.indexOf("the") + ",");  
document.write(str.indexOf("the",7));
```

This example gives the following output:

6,16

Since the search is case-sensitive, the index of the first match is not 0 but 6. If the search were to start at index 7, then the second lowercase "the" would be found (index 16).

### 7.11.10 lastIndexOf method

This method searches the calling string from beginning to end for the string specified in *searchValue*. If *fromIndex* is specified, then the search starts at this index. Otherwise, it starts at *string.length - 1*. While searching, the method distinguishes between uppercase and lowercase

---

```
lastIndexOf(searchValue)  
lastIndexOf(searchValue, fromIndex)
```

---

#### Return value

Index of the first character of the match closest to the end of the string, or -1 if *searchValue* is not found

#### Parameters

*searchValue*

String to be searched for

*fromIndex*

Position in the calling string object at which the search for the string specified in *searchValue* is to begin

Default: *string.length - 1*

#### Example

```
str = "The a the b and the c.";
document.write(str.lastIndexOf("the") + ",");
document.write(str.lastIndexOf("the",6) + ",");
document.write(str.lastIndexOf("the",5));
```

This example gives the following output:

```
16,6,-1
```

The index of the match closest to the end is 16.

6 is the smallest index at which the string "the" is found when searching from the right. This illustrates the distinction between uppercase and lowercase.

### 7.11.11 match method

This method searches the calling string for the search pattern specified in *pattern*.

If the expression *pattern* is a `RegExp` object, then matching substrings are searched for in the calling string. Otherwise, *pattern* is converted to type `string`. In this case, the second parameter *mode* (of type `string`) is taken into account.

If *pattern* is of type `string` and if *mode* contains the character `g`, or if *pattern* is a regular expression with the `g` flag (global search), then each substring of the calling string which matches the pattern is entered in precisely one element of the result array. Otherwise, `match` operates like the `exec` method for regular expressions, i.e. only the first match is considered.

---

```
match(pattern)
match(pattern, mode)
```

---

#### Return value

If a substring matching *pattern* is found in the calling string, a result array is returned in the format described above. Otherwise, the value `null` is returned.

#### Parameters

*pattern*

A search pattern.

*pattern* can be an expression of type `string` or a `RegExp` object (see [section "RegExp class" on page 180](#)).

*mode*

Expression of type `string` which specifies the mode (this type of mode specification is only taken into account if *pattern* is not a `RegExp` object):

`i` (ignoreCase) Ignore uppercase/lowercase.

`g` (global) Find all suitable matches.

`ig` or `gi`

Ignore uppercase/lowercase and find all suitable matches.

*Example 1 (with g flag)*

```
str="Gabi, Alice, A,Karla, another, Andreas, Martin, Andy, Pit";
re=/A(\w+)/g;
resultArray=str.match(re);

for (attr in resultArray)
  document.write("resultArray." + attr + ": " + resultArray[attr] + "<BR>");

for (attr in re)
  document.write("re."+attr + ": " + re[attr] + "<BR>");
```

This example gives the following output:

```
resultArray.0: Alice
resultArray.1: Andreas
resultArray.2: Anke
re.global: true
re.ignoreCase: false
re.lastIndex: 0
re.source: (A(\w+))
```

### *Example 2 (without g flag)*

```
str="Gabi, Alice, A,Karla, another, Andreas, Martin, Andy, Pit";
re=/A(\w+)/;
resultArray=str.match(re);

for (attr in resultArray)
  document.write("resultArray." + attr + ": " + resultArray[attr] + "<BR>");

for (attr in re)
  document.write("re."+attr + ": " + re[attr] + "<BR>");

for (attr in RegExp)
  document.write("RegExp." + attr + ": " + RegExp[attr] + "<BR>");
```

This example gives the following output:

```
resultArray.0: Alice
resultArray.1: Alice
resultArray.2: llice
resultArray.index: 6
resultArray.input: Gabi, Alice, A,Karla, another, Andreas, Martin, Anke, Pit
re.global: false
re.ignoreCase: false
re.lastIndex: 12
re.source: (A(\w+))
RegExp.$1: Alice
RegExp.$2: llice
RegExp.input:
RegExp.lastMatch: Alice
RegExp.lastParen: llice
RegExp.leftContext: Gabi,
RegExp.multiline: false
RegExp.rightContext: , A,Karla, another, Andreas, Martin, Anke, Pit
```



### 7.11.12 replace method

This method replaces substrings of the calling string with the result of the *ReplaceString* expression. The modified string is returned, while the calling string itself remains unchanged.

---

```
replace(Pattern, ReplaceString)
```

---

#### Return value

Modified string in which substrings of the calling string have been replaced as specified

#### Parameters

##### *Pattern*

Expression of type `string` or `RegExp` object, which specifies a pattern for the substrings that are to be replaced

##### *ReplaceString*

Replacement string that may contain the following \$ variables, which are determined from the matching substrings for replacement and their context:

\$&	Entire matching string
\$+	Match for last bracketed expression
\$1, \$2, ..., \$9	Match for 1st, 2nd, ... 9th bracketed expression
\$`	Left-hand context
\$'	Right-hand context

##### *Example 1*

```
re = /monday/gi;  
str = "Monday morning and monday evening";  
newstr = str.replace(re, "tuesday");  
document.write(newstr);
```

The specification of the regular literal `re` sets the flags `g` (global) and `i` (ignoreCase). In this example, both flags are necessary in order to replace the Mondays. This example gives the following output: "tuesday morning and tuesday evening".

*Example 2*

```
re = /(\w+)\s(\w+)/;
str = "today not!John Smith";
newstr = str.replace(re, "$2 $1");
document.write(newstr);
```

A literal is specified for the regular expression `re`. This searches for the first space character located between two character sequences each of which contains at least one alphanumeric character. Thanks to the use of the brackets, the character sequence to the left of the space ("today") is stored in property `$1` while the character sequence to the right of it ("not") is stored in `$2`.

This example gives the following output: "not today!John Smith".

### 7.11.13 search method

This method searches the calling string for a regular expression

---

```
search(pattern)
```

---

#### Return value

Position of the first matching substring, or -1 if no match is found

#### Parameters

*pattern*

Regular expression to be searched for in the calling string. *pattern* can be an expression of type `string` or a `RegExp` object (see [section “RegExp class” on page 180](#)).



If you simply want to know whether or not a particular character combination occurs in a string, use the `search` method. Although the `match()` method provides more information, it also takes longer to execute.

#### Example

```
function check(str,re) {
  if(str.search(re) != -1)
    result=" contains the pattern: ";
  else
    result=" does not contain the pattern: ";
  document.write(str+result+re.source);
}
check("WebTransactions",/bt/i);
```

This call checks whether `WebTransactions` contains a match for the regular expression `/bt/i`. Since the `i` flag is set, uppercase/lowercase notation is ignored and a match is found. The output also indicates that the specified flags are not present in the `re.source` property:

```
WebTransactions contains the pattern: bt
```

### 7.11.14 setValue method

This method assigns a new value to the calling string object.

---

```
setValue(value)
```

---

#### **Return value**

None

#### **Parameters**

*value* New value for the calling string object

## 7.11.15 slice method

This method returns the section of the calling string which lies between the two specified indices.

---

```
slice()  
slice(indexA)  
slice(indexA, indexB)
```

---

### Return value

Section of the calling string which lies between the two specified indices (including the character specified by the smaller index but excluding the character specified by the larger index)

### Parameters

*indexA* Expression specifying the index for the start of the substring

If *indexA* is less than 0, the value 0 is assumed.

If *indexA* is greater than *indexB*, an empty string is returned.

If *indexA* is greater than *callingString.length*, an empty string is returned.

Default: 0

*indexB*

Expression specifying the index for the end of the substring

If *indexB* is less than 0, it specifies the offset from the end of the string (*callingString.length+indexB*).

If *indexB* is greater than *callingString.length*, the value *callingString.length* is assumed.

Default: *callingString.length*

If *indexB* is not specified, the value *callingString.length* is assumed for *indexB*, i.e. the substring from *indexA* through to the end of *callingString* is returned.

If both indices are identical, an empty string is returned.

### Example

```
document.write("<BR>" + "WebTransactions".slice(0,3));  
document.write("<BR>" + "WebTransactions".slice(4,3));  
document.write("<BR>" + "WebTransactions".slice(3));  
document.write("<BR>" + "WebTransactions".slice(0,-12));
```

This example gives the following output:

```
Web
Transactions
Web
```

### 7.11.16 split method

This method returns an array of substrings of the calling string.

---

```
split()
split(pattern)
split(pattern, limit)
```

---

#### Return value

Result array containing elements formed from the set of characters from the start to the first separator, between each pair of separators, and between the last separator and the end of the string. The separators themselves are not present in the result.

#### Parameters

*pattern* Object of type `RegExp` or `String`. *pattern* specifies a pattern for a separator.

If *pattern* is not specified, `split` returns a single entry for the entire string.

If the separator is an empty string, `split` returns a separate entry for each character in the calling string.

*limit* The maximum number of separators

#### Example

```
numberString = ";7.1;687.1;32.634;.56;";
document.write ("input string: " + numberString + "<BR>" );
numberArray = numberString.split(";");
document.write("resulting array: |");
for (i in numberArray)
    document.write(numberArray[i] + "|");
```

The `split` call removes the separating semi-colons and organizes the numbers in an array. This example gives the following output:

```
input string: ;7.1;687.1;32.634;.56;
resulting array: ||7.1|687.1|32.634|.56||
```

### 7.11.17 substr method

This method returns the section of the calling string which begins at the specified index and has the specified length.

---

```
substr(index)  
substr(index,length)
```

---

#### Return value

Section of the calling string which begins at the specified index (including the character specified by the index) and has the specified length

#### Parameters

*index* Expression specifying the index for the start of the substring

If *index* is less than 0, it specifies the offset from the end of the string (*callingString.length+index*). If this offset is located before the start of the calling string, the value 0 is assumed.

If *index* is greater than or equal to *callingString.length*, an empty string is returned.

*length* Expression specifying the length of the substring

If *length* is not specified, the substring from the *index* to the end of *callingString* is returned.

If *length* is less than or equal to 0, an empty string is returned.

#### Example

```
document.write("<BR>" + "WebTransactions".substr(3,5));  
document.write("<BR>" + "WebTransactions".substr(4,0));  
document.write("<BR>" + "WebTransactions".substr(3));  
document.write("<BR>" + "WebTransactions".substr(-12,2));
```

This example gives the following output:

Trans

Transactions

Tr

### 7.11.18 substring method

This method returns the section of the calling string which lies between the two specified indices.

---

```
substring(indexA)  
substring(indexA, indexB)
```

---

#### Return value

Section of the calling string which lies between the two specified indices (including the character specified by the smaller index but excluding the character specified by the larger index)

#### Parameters

*indexA*, *indexB*

Expressions specifying indices

If *indexB* is not specified then the value `callingString.length` is assumed for *indexB*, i.e. the substring from *indexA* to the end of *callingString* is returned.

If an index lower than 0 is specified, then the value 0 is assumed. If an index greater than or equal to `callingString.length` is specified, then `callingString.length` is assumed.

If the two indices are identical, an empty string is returned. If *indexA* is greater than *indexB*, the two arguments are swapped, and the substring in the middle is returned.

#### Example

```
document.writeln("<BR>" + "WebTransactions".substring(0,3));  
document.writeln("<BR>" + "WebTransactions".substring(99,3));  
document.writeln("<BR>" + "WebTransactions".substring(0));  
document.writeln("<BR>" + "WebTransactions".substring(-99)+"!!!");
```

This example gives the following output:

```
Web  
Transactions  
WebTransactions  
WebTransactions!!!
```



### 7.11.19 toLowerCase method

This method returns the value of the calling string with all uppercase characters (A–Z) converted to the corresponding lowercase characters. The calling string is not modified.

---

```
toLowerCase()
```

---

#### Return value

String generated from the calling string by converting all uppercase characters (A–Z) to the corresponding lowercase characters

#### Parameters

None

#### Example

```
a="DaDa";  
b=a.toLowerCase();  
document.write(b+a);
```

This example gives the following output:

```
dadaDaDa
```

### 7.11.20 toString method

This method returns the value of the calling object in string format.

---

```
toString()
```

---

#### Return value

Value of type `string`

#### Parameters

None

### 7.11.21 toUpperCase method

This method returns the value of the calling string with all lowercase characters (a-z) converted to the corresponding uppercase characters. The calling string is not modified.

---

```
toUpperCase()
```

---

#### Return value

String generated from the calling string by converting all lowercase characters (a-z) to the corresponding uppercase characters

#### Parameters

None

#### Example

```
a="DaDa";  
b=a.toUpperCase();  
document.write(b+a);
```

This example gives the following output:

```
DADADaDa
```

### 7.11.22 valueOf method

This method returns the value of the calling object in string format.

---

```
valueOf()
```

---

#### Return value

Value of type `string` which represents the value of the calling object

#### Parameters

None

## 7.12 WT\_Communication class

Objects belonging to the `WT_Communication` class are known as communication objects. These communication objects allow you to handle parallel connections and thus integrate multiple host applications within a single WebTransactions application. The communication objects contain information about the communication module in use, the current connection status, the data most recently received from the host application, etc. (for more information, refer to the WebTransactions manual “Concepts and Functions”).

### 7.12.1 Constructors

The constructors create a new communications object. The return value is a reference to this object which can be used when the object has to be referenced again.

---

```
WT_Communication()  
WT_Communication(handle)
```

---

#### Return value

Communication object of the `WT_Communication` class

#### Parameters

*handle*

Name of the communication object. WebTransactions creates the communication object in the host root object `WT_HOST`. In addition, a connection-specific system object is created in this newly created communication object and the attributes of this system object are used to control the connection in question.

If the constructor is called without an argument, WebTransactions determines the name from the global system object's `HANDLE` attribute. If this is also not set, then an error occurs.

### 7.12.2 close method

This method terminates the connection to the host application but does not delete the communication object: if followed by an `open` call, the connection can then be reopened. Communication objects survive until the end of the session.

---

`close()`

---

**Return value**

None

**Parameters**

None

### 7.12.3 equals method

This method compares the calling `WT_Communication` object with the object transferred in the *object* argument for equality of class, attributes, and values.

---

`equals(object)`

---

**Return value**Boolean return value: `true` if the two objects are equal, `false` if not**Parameters***object* Object with which the calling object is to be compared

### 7.12.4 `getClassName` method

This method returns a string specifying the name of the class to which the calling object belongs.

---

```
getClassName()
```

---

#### **Return value**

String specifying the class to which the calling object belongs, in this case "WT\_Communication".

#### **Parameters**

None

### 7.12.5 `getModule` method

This method returns the name of the communication module in a string.

---

```
getModule()
```

---

#### **Return value**

String containing the name of the communication module

#### **Parameters**

None

### 7.12.6 open method

The method establishes a connection to a host application. To call the `open()` method, the associated communication object must already exist.

If the action returns an error code, this is stored as an `ERROR` attribute in the global system object.

---

```
open()  
open(protocol)
```

---

#### Return value

Reference to an object of the `WT_Communication` class

#### Parameters

*protocol*

Allows you to select one of the communication modules linked to WebTransactions. If you call `open` without any argument, WebTransactions determines the communication module from the global system object's `PROTOCOL` attribute. If this is also not set, WebTransactions reports an error.

### 7.12.7 receive method

This method receives a message from the host and stores it in the calling object. It is then possible to access the stored data via the host data objects.

---

```
receive()
```

---

#### Return value

The calling object if the function is successful, or `null` otherwise. A corresponding error message is stored in the global system object's `ERROR` attribute.

#### Parameters

None

### 7.12.8 send method

This method sends the message stored in the host data objects of the calling communication object to the host.

---

```
send()
```

---

#### Return value

The calling object if the function is successful, or `null` otherwise. A corresponding error message is stored in the global system object's `ERROR` attribute.

#### Parameters

None

## 7.13 WT\_Filter class

The `WT_Filter` class is a universal filter task used for the following tasks:

- The portable representation of data for communication with external applications via XML messages (XML=eXtended Markup Language).

This allows you to convert data from WTScrip programs into XML documents for further processing with external applications, and to process data supplied from external sources in the form of XML documents in WebTransactions.

This is described in detail in [section “Importing and exporting XML documents” on page 367](#).

- The conversion of WTScrip data structures into XML documents and vice versa.  
For instance, this allows you to receive internal data even after the end of a session, which is converted by calling `WT_Filter` methods and stored using `document` methods.

This is described in detail in [section “Exporting data structures” on page 372](#).

- The conversion of WTScrip method calls into XML documents and vice versa in order to allow communication between WebTransactions applications.

This is described in detail in the WebTransactions manual “Client APIs for WebTransactions”.

- Processing of any XML document with freely-selectable callback functions. An example is given on [page 230](#).

The methods described below are also available as class methods.



### 7.13.1 dataObjectToXML method

This method generates an XML document from a WTScripT data object. The generated document has the XML format described in [section “Exporting data structures” on page 372](#).

---

```
dataObjectToXML(objectPattern)
dataObjectToXML(objectPattern, objectName, ...)
```

---

#### Return value

XML document generated from a WTML script

#### Parameters

*objectPattern*

WTScripT data from whose structures an XML document is to be generated and returned.

*object*

Any object.

The data object *object* and all its attributes. If these attributes are themselves objects, conversion takes place recursively for these objects.

*object.*

Any object without attributes.

The data object *object*. The terminating period prevents the attributes of this object from being converted.

*object..*

Any object but not its subordinate objects.

The data object *object* and all its attributes (since nothing is specified between the two subsequent periods). The terminating period prevents the subattributes of subordinate objects under *object* from being converted.

*object..value*

All attributes with the same name located on the level below an object.

All attributes with the name *value* which are contained in objects directly underneath the data object *object*.

*object..val1|val2*

Several attributes with the same name located on the level below an object.

All attributes with the name *val1* or *val2* which are contained in objects directly underneath the data object *object*.

If you want to convert the calling object within a method, then you should not use the "this" string as the pattern. However, you can use a local variable to point to the calling object and then use this variable:

```
dummy1 = this;  
text=dataObjectToXML("dummy1");
```

In the same way, an XML document can be unpacked for the calling object:

```
dummy2=this;  
XMLToDataObject(text,"dummy2");
```

*objectName*

(Optional) This argument allows you to override the top-level name.

Object methods will not be serialized with `WT_Filter.dataObjectToXML`. The names will be serialized and restored but it will not be possible to call them (they will be indicated as obsolete). `WT_Filter.dataObjectToXML` and `WT_Filter.XMLToDataObject` are designed for data objects only.

### 7.13.2 dataObjectToFormattedXML method

This method creates a formatted XML document from a WTScript data object in accordance with the XML representation described in the [section “Exporting data structures” on page 372](#).

---

```
dataObjectToFormattedXML(objectPattern, format)
dataObjectToFormattedXML(objectPattern, format, objectName, ...)
```

---

#### Return value

XML document created from a WTScript.

#### Parameters

*objectPattern*

This argument specifies the names of WTScript data from whose structures an XML document is to be created and returned.

*object*

Any object.

The data object *object* and all its attributes. If attributes are in turn objects, conversion is performed recursively for these objects.

*object.*

Any object without attributes.

The data object *object*. The final dot, however, ensures that no attributes of this object are converted.

*object..*

Any object without child objects.

The data object *object* and all the attributes of this data object (because no specification was made between the dots). The final dot ensures that no child attributes of child objects in *object* are converted.

*object..value*

All matching attributes of one level within an object.

All objects with the name *value* contained in objects directly below the data object *object*.

*object..value1|value2*

Multiple matching attributes one level below an object.

All objects with the name *value1* or *value2* contained in objects directly below the data object *object*.

If you wish to convert the calling object within a method, you must not specify the string "this" as the pattern. You can, however, point a local variable to the calling object and use this:

```
dummy1 = this;
text=dataObjectToXML("dummy1");
```

An XML document can be unpacked to the calling object in the same way:

```
dummy2=this;
XMLToDataObject(text,"dummy2");
```

### *format*

String with format specifications. No distinction is drawn between uppercase and lowercase.

WebTransactions V7.5 currently only supports the value `NL` for *format*:

<code>NL</code>	A new line is additionally created after the end of each tag, i.e. after the closing angle bracket (>). This improves readability of the XML output and allows different versions to be compared.
-----------------	---

If you do not specify *format*, the XML output is made up of a single line.

### *objectName*

(optional) This argument allows the top-level names to be overwritten.

Methods of objects are not serialized with `WT_Filter.dataObjectToXML`. Only the names are serialized and restoration creates methods that cannot be called (known as 'obsolete').

`WT_Filter.dataObjectToXML` and `WT_Filter.XMLToDataObject` are only intended to be used with data objects.

### *Example*

```
o=new Object();
o.s='abc'; o.n=42; o.b=true;
o.S=new String('ABC'); o.N=new Number(43); o.B=new Boolean(false);
o.D=new Date();
o.u=a; // a is not defined
o.nu=null;
o.O={n:1, m:2};
o.A=[1,2,,3,,];
xml=WT_Filter.dataObjectToFormattedXML('o','n1');
```

This example generates the following output:

```
<data>
<object name="o" class="Object"><object name="A" class="Array"><number
name="0">1</number>
<number name="1">2</number>
<number name="3">3</number>
</object>
<object name="B" class="Boolean">false</object>
<object name="D" class="Date">1275668587840</object>
<object name="N" class="Number">43</object>
<object name="O" class="Object"><number name="m">2</number>
<number name="n">1</number>
</object>
<object name="S" class="String">ABC</object>
<boolean name="b">true</boolean>
<number name="n">42</number>
<object name="nu" class="Undefined"></object>
<string name="s">abc</string>
<undefined name="u"/></object>
</data>
```

### 7.13.3 methodCallToXML method

This method generates an XML document for calling a method. It is used to prepare a WebTransactions method call of another WebTransactions application for transmission to the WT\_REMOTE interface (for more information, see the WebTransactions manual “Client APIs for WebTransactions”).

---

```
methodCallToXML(methodName)  
methodCallToXML(methodName, argArray)  
methodCallToXML(methodName, argArray, codeBase)
```

---

#### Return value

String containing the XML document for calling a method

#### Parameters

*methodName*

Name of a method

*argArray*

Array with the arguments for the method (if required)

*codeBase*

WTML document containing the function definition in the remote WebTransactions application

### 7.13.4 objectTreeToXML method

This method requires you to specify a WTSript data structure that complies with certain conventions, and converts this into an XML document which is returned as the result.

---

`objectTreeToXML(xmlObject)`

---

#### Return value

XML document generated on the basis of the specified WTSript data structure

#### Parameters

*xmlObject*

WTSript data structure, as described in [section “Exporting data structures” on page 372](#). The `child` attribute may be omitted here.

If *xmlobject* contains syntax errors, an error message is output. In this case, the method returns an XML document which has been generated as far as possible.

### 7.13.5 XMLToDataObject method

This method performs the reverse operation to `dataObjectToXML` (described in [section “dataObjectToXML method” on page 217](#)). It generates a WTScript data structure from XML text (see also [section “Exporting data structures” on page 372](#))

This method is additive. This means that if the XML serialized object already exists, it will not be completely replaced by `WT_Filter.XMLToDataObject`, instead the attribute contained in the XML document will be extended. Previously existing attributes (and methods) will remain unchanged.

---

```
XMLToDataObject(xmlObjectText[, suppressWhitespace])
XMLToDataObject(xmlObjectText, objectName, ...[, suppressWhitespace])
```

---

#### Return value

There is no return value in the strict sense of the word. The function creates an object which is saved in `xmlObjectText` (where possible under the name `objectName`).

#### Parameters

*xmlObjectText*

XML text that describes a WTScript data object

*objectName*

(Optional) Allows you to override the top-level name

*suppressWhitespace*

(optional) Handling of the XML document free data. This attribute will be converted to the `boolean` type if necessary. If this returns `true`, the leading and trailing whitespace characters of the data string will be suppressed.

The default value is `false`.



If you want to convert the calling object within a method, then you should not use the "this" string as the pattern. However, you can use a local variable to point to the calling object and then use this variable:

```
dummy1 = this;
text=dataObjectToXML("dummy1");
```

In the same way, an XML document can be unpacked for the calling object:

```
dummy2=this;
XMLToDataObject(text,"dummy2");
```



### 7.13.6 XMLToMethodCall method

This method interprets an XML document that describes a method call and executes it if necessary (for more information, see the WebTransactions manual “Client APIs for WebTransactions”).

---

`XMLToMethodCall(xmlInvokeText)`

---

#### Return value

Return value of the function described in the interpreted XML document

#### Parameters

*xmlInvokeText*

This argument is converted to type `string` and interpreted as an XML document. If the document contains a method call, the method is sought in the specified WTML document and executed. The parameters specified in the reverse method `methodCallToXML` are also taken into consideration here.

### 7.13.7 XMLToObjectTree method

This method performs the reverse operation to `objectTreeToXML` (described in [section “objectTreeToXML method” on page 223](#)). It requires you to enter a syntactically correct XML text and transforms this into a corresponding WTSript data structure which is returned as the result (see also [section “Importing and exporting XML documents” on page 367](#)).

---

```
XMLToObjectTree(xmlText[, suppressWhitespace])
```

---

#### Return value

WTSript data structure generated on the basis of the specified XML text

#### Parameters

*xmlText*

Syntactically correct XML text in string format

*suppressWhitespace*

(optional) Handling of the XML document free data. This attribute will be converted to the `boolean` type if necessary. If this returns `true`, the leading and trailing whitespace characters of the data string will be suppressed.

The default value is `false`.

If *xmlText* contains syntax errors, XML entities, or processing statements, an error message is output. In this case, the method returns only a fragment of a WTSript data structure which matches the semantics of the XML text as far as possible.

### 7.13.8 Methode XML\_SAXParse

This method enables an XML document to be interpreted. This method analyses the XML document and starts a corresponding callback function when recognizing start or end tags, data or processing statements. These callback functions must be made available in the form of WTScript functions or function objects. The functions are transferred to the method by means of a handler object.

---

XML\_SAXParse (*xmlDocument*, *HandlerObject* [, *suppressWhitespace* ])

---

#### Return value

Return value of the integrated XML parsers as a type object with the ErrorCode attributes of the number type and ErrorText of the string type with the following contents:

ErrorCode	ErrorText	Meaning
0	XML_ERROR_NONE	Everything OK
1	XML_ERROR_NO_MEMORY	No more memory available
2	XML_ERROR_SYNTAX	Syntax fault in document
3	XML_ERROR_NO_ELEMENTS	No element found
4	XML_ERROR_INVALID_TOKEN	Not well formed
5	XML_ERROR_UNCLOSED_TOKEN	No closed token
6	XML_ERROR_PARTIAL_CHAR	No closed token
7	XML_ERROR_TAG_MISMATCH	Tag mismatch
8	XML_ERROR_DUPLICATE_ATTRIBUTE	Double attribute
9	XML_ERROR_JUNK_AFTER_DOC_ELEMENT	Invalid item after document element
10	XML_ERROR_PARAM_ENTITY_REF	Impermissible reference to entity
11	XML_ERROR_UNDEFINED_ENTITY	Undefined entity
12	XML_ERROR_RECURSIVE_ENTITY_REF	Recursive entity reference
13	XML_ERROR_ASYNC_ENTITY	Asynchronous entity
14	XML_ERROR_BAD_CHAR_REF	Reference to invalid character
15	XML_ERROR_BINARY_ENTITY_REF	Reference to binary entity
16	XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF	Reference to external entity in attribute
17	XML_ERROR_MISPLACED_XML_PI	Processing statements not at beginning of an external entity
18	XML_ERROR_UNKNOWN_ENCODING	Unknown encoding
19	XML_ERROR_INCORRECT_ENCODING	Defined encoding is incorrect

ErrorCode	ErrorText	Meaning
20	XML_ERROR_UNCLOSED_CDATA_SECTION	CDATA not closed.
21	XML_ERROR_EXTERNAL_ENTITY_HANDLING	Fault in processing of an external entity

## Parameter

### *xmlDocument*

XML-Dokument to be interpreted. If this is an object of the document class, the content of the corresponding file is interpreted. Every other type is if necessary converted to `string` type and is then interpreted.

### *HandlerObject*

Declaration of callback functions. This is an object of the object class and has the following layout:

`StartElementHandler`:

reference to the callback function that processes the start tags.

`EndElementHandler`:

reference to the callback function that processes the end tags.

`CharacterDataHandler`:

reference to the callback function that processes the data.

`ProcessingInstructionHandler`:

reference to the callback function that processes the processing statements

The attributes are optional. If they are not defined or if they do not reference a function or a function object, no corresponding handler is started.

### *suppressWhitespace*

(optional) processing of the free data of the XML document. This attribute is if necessary converted to `boolean` type. If this returns `true`, all the leading and concluding Whitespace characters of the data string are suppressed.

Default value is `false`.

### Prototype of single callback function

The callback routines for the XML parser must have the following form. Validity is not tested.

```
StartTagHandler ( string tagname, Object[] nameValue[], string namespace )
```

*tagname*

The name of the opened tag is transferred here (**without** namespace prefix).

*nameValue*

In this array the name value pairs of the tag attributes are transferred as single objects. The attribute names of the objects are *name* und *value*.

*namespace*

Contains the namespace the tag belongs to or an empty string.

```
EndTagHandler ( string tagname[], string namespace )
```

*tagname*

The name of the closing tag is transferred here (**without** namespace prefix).

*namespace*

The name of any valid namespace or empty string is transferred here.

```
CharacterDataHandler ( string data )
```

*data* Free data is transferred here.

```
ProcessingInstructionHandler ( string target, string data )
```

*target* The name of the processing statement is transferred as a string here.

*data* The found processing statement is transferred as a string here.

*Example*

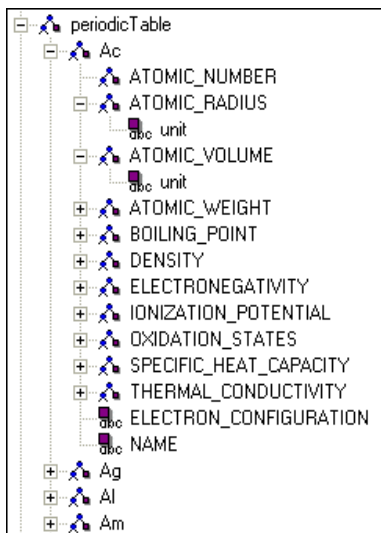
The basis for the following examples is the XML file `allelements.xml`. This file contains a list of all chemical elements and their features.

```
<PERIODIC_TABLE>
  <ATOM>
    <NAME>Actinium</NAME>
    <ATOMIC_WEIGHT>227</ATOMIC_WEIGHT>
    <ATOMIC_NUMBER>89</ATOMIC_NUMBER>
    <OXIDATION_STATES>3</OXIDATION_STATES>
    <BOILING_POINT UNITS="Kelvin">3470</BOILING_POINT>
    <SYMBOL>Ac</SYMBOL>
    <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
      10.07
    </DENSITY>
    <ELECTRON_CONFIGURATION>[Rn] 6d1 7s2 </ELECTRON_CONFIGURATION>
    <ELECTRONEGATIVITY>1.1</ELECTRONEGATIVITY>
    <ATOMIC_RADIUS UNITS="Angstroms">1.88</ATOMIC_RADIUS>
    <ATOMIC_VOLUME UNITS="cubic centimeters/mole">
      22.5
    </ATOMIC_VOLUME>
    <SPECIFIC_HEAT_CAPACITY UNITS="Joules/gram/degree Kelvin">
      0.12
    </SPECIFIC_HEAT_CAPACITY>
    <IONIZATION_POTENTIAL>5.17</IONIZATION_POTENTIAL>

    <THERMAL_CONDUCTIVITY UNITS="Watts/meter/degree Kelvin">
    <!-- At 300K -->
      12
    </THERMAL_CONDUCTIVITY>
  </ATOM>

  <ATOM>
    <NAME>Aluminum</NAME>
    ...
  </ATOM>
</PERIODIC_TABLE>
```

The following document has to be read and transformed into a WTScript data structure that contains an attribute for each element with the symbol of the element:



The following function is called when an opening XML tag is captured. It stores the attributes of the tag in the global variable `CurrentAttributes`, so that the variable is available in the end tag handler. If a new description of a chemical element begins a new WTScript object is generated that represents this description.

```
function StartChemie (name,attributes)
{
    CurrentAttributes = attributes;
    CurrentData = "";
    if (name == "ATOM")
        CurrentObject = new Object();
}
```

The following function is called if data has been found. Stores data for the end tag handler:

```
function DataChemie (data)
{
    CurrentData = data;
}
```

The following function is called when a closing XML tag is recognized. It executes different actions depending on the name of the closing tag.

- If SYMBOL is recognized the name is temporarily stored from the relevant data and is used as an attribute name in the WTScript data structure.
- If ATOM is recognized, the description of a chemical element is complete and can be stored under the symbol name.
- All other elements are entered under the tag name into the object that describes the chemical element.
- If the current tag describes numerical data, the unit of the data may have been specified in the UNIT attribute. This information is supplied by the attributes temporarily stored in the start tag handler and is entered into the data structure.

```
function EndChemie (name)
{
  switch (name)
  {
    case "SYMBOL":
      CurrentName = CurrentData;
      break;
    case "ATOM":
      //End of element description: add whole element to WTScript
      // object
      periodicTable[CurrentName] = CurrentObject;
      delete CurrentObject;
      break;
    case "NAME":
    case "ELECTRON_CONFIGURATION":
      CurrentObject[name] = CurrentData;
      break;
    case "PERIODIC_TABLE": //end of description: cleanup
      delete CurrentName;
      delete CurrentData;
      delete CurrentObject;
      break;
    default:
      //All the other information about a chemical element contain
      // numerical data and may contain the attribut UNIT, so create
      // as Number object and add the
      // attribute unit to this object if necessary
      CurrentObject[name] = new Number(CurrentData);
      if (CurrentAttributes.length == 1
          && CurrentAttributes[0].name == "UNITS")
        CurrentObject[name].unit = CurrentAttributes[0].value;
  }
}
```



```
}

// Create a document object for the XML file
doc = new Document(WT_SYSTEM.BASEDIR + "/allelements.xml" );

// Prepare the handler document
handlerObject = new Object();
handlerObject.StartElementHandler = Start chemical element;
handlerObject.EndElementHandler = End chemical element;
handlerObject.CharacterDataHandler = Data chemical element;

//Data structure for information from the XML file. The element handler enters
the information here.
periodicTable = new Object();

//Call filter
try {
    WT_Filter.XML_SAXParse( doc, handlerObject,true );
}
catch (exc) {
    document.writeln(exc);
    exitTemplate();
}
```

## 7.14 WT\_LdapConnection class

In order to ensure support for the Internet communication protocol LDAP (**L**ightweight **D**irectory **A**ccess **P**rotocol) in WebTransactions, the `WT_LdapConnection` class has been defined in `WTScript`. `WT_LdapConnection` only supports synchronous operations, and does not allow for extended operations.

This section contains the following information:

- an overview of the LDAP directory service
- the LDAP error messages
- built-in methods of the `WT_LdapConnection` class
- examples

### 7.14.1 Overview of the LDAP directory service

This section is based on the "SLAPD and SLURPD Administrators Guide" published by the University of Michigan, and is intended to provide a brief overview of the LDAP directory service. For further information on LDAP in BS2000/OSD, please refer to the "[interNet Services](#)" Administrators Guide.

#### What is a directory service?

A directory is like a database, but tends to contain more descriptive, attribute-based information. The information in a directory is generally read more often than it is written. As a consequence, directories don't usually implement the complicated transaction or roll-back schemes regular databases use for doing high-volume complex updates. Directory updates are typically simple all-or-nothing changes, if they are allowed at all. Directories are tuned to give quick responses to high-volume lookup or search operations. They may have the ability to replicate information widely in order to increase availability and reliability, while reducing response time. When directory information is replicated, temporary inconsistencies between the replicas may be OK, as long as they get in sync eventually.

There are many different ways to provide a directory service. Different methods allow different kinds of information to be stored in the directory, place different requirements on how that information can be referenced, queried, and updated, how it is protected from unauthorized access etc. Some directory services are local, providing service to a restricted context (e.g. the finger service on a single machine). Other services are global, providing service to a much broader context (e.g. the entire Internet). Global services are usually distributed, meaning that the data they contain is spread across many machines, all of which cooperate to provide the directory service. Typically a global service defines a uniform namespace which gives the same view of the data no matter where you are in relation to the data itself, i.e. it always responds to a request with the same result irrespective of the machine on which the request was issued.

## What is LDAP and how does it work?

LDAP is a directory service protocol that runs over TCP/IP. The nitty-gritty details of LDAP are defined in RFC 1777 "The Lightweight Directory Access Protocol".

The LDAP directory service is based on a client/server model. One or more LDAP servers contain the data making up the LDAP directory tree. An LDAP client connects to an LDAP server and asks it a question. The server responds with the answer, or with a pointer to where the client can get more information (typically, another LDAP server). No matter which LDAP server a client connects to, it sees the same view of the directory; a name presented to one LDAP server references the same entry it would at another LDAP server. This is an important feature of a global directory service, like LDAP.

The LDAP directory service model is based on entries. An entry is a collection of attributes that has a name, called a distinguished name (DN). The DN is used to refer to the entry unambiguously. Each of the entry's attributes has a type and one or more values. The types are typically mnemonic strings, like *cn* for common name, or *mail* for email address. The values depend on what type of attribute it is. For example, a *mail* attribute might contain the value *babs@umich.edu*. A *jpegPhoto* attribute would contain a photograph in binary JPEG/JFIF format.

In LDAP, directory entries are arranged in a hierarchical tree-like structure that reflects political, geographic, and/or organizational boundaries. Entries representing countries appear at the top of the tree. Below them are entries representing states or national organizations. Below them might be entries representing people, organizational units, printers, documents, or just about anything else you can think of.

### 7.14.2 LDAP error messages

Message number	Message text
<b>0300</b>	0300:Error: LDAP(%d) - %s
<b>0301</b>	0301:Error: LDAP - invalid arguments.
<b>0302</b>	0302:Error: LDAP initialization - cannot initialize a LDAP session.
<b>0303</b>	0303:Error: LDAP handle - handle not found.

Error handling in WTSript is described in the WebTransactions manual "Concepts and Functions". An example of the handling of an `LdapError` exception can be found on [page 257](#).

### 7.14.3 Constructor

The constructor of the `WT_LdapConnection` class creates an LDAP object, and opens a connection with the host specified in *hostname* and the port number specified in *port*.

---

```
WT_LdapConnection()  
WT_LdapConnection(hostname)  
WT_LdapConnection(hostname, port)
```

---

#### Return value

Reference to the LDAP object created or NULL

#### Parameters

*hostname*

Name of the connection host  
Default: "localhost"

*port*

Port number of the connection  
Default: 389

#### Example

```
L=new WT_LdapConnection("localhost");  
// Execute LDAP operations  
delete L;
```

### 7.14.4 add method

This method adds entries to the LDAP directory. Its parameters include the distinguished name of the new entry (*dn*) and an array containing its attributes. The attribute values can be specified in string format or in the form of arrays.

---

`add(dn, entry)`

---

#### Return value

None

#### Parameters

*dn*      Distinguished name (DN) of the new entry

*entry*    Information on the entry

#### Example

```
dn="cn= Harry G. Miller, ou=board, o=FTS, c=DE";
entry= new object();
entry.sn="Harry";
entry.mail=new Array("Harry@my_company.net",
                    "HGMiller@my_company.net",
                    "NNN@my_company.net");
entry.objectclass="Person";
L=new WT_LdapConnection("localhost");
L.bind();
L.add(dn, entry);
L.unbind();
delete L;
```

### 7.14.5 bind method

This method opens an LDAP session. The *bind\_rdn* and *bind\_password* parameters are used to control access to the LDAP directory. If a `bind` call is issued with no argument specified for these two optional parameters, access will be anonymous. Only LDAP V3 is supported.

---

```
bind()  
bind(bind_rdn)  
bind(bind_rdn, bind_password)
```

---

#### Return value

None

#### Parameters

*bind\_rdn*

User ID

Default: anonymous user ID

*bind\_password*

Password

#### Example

```
L=new WT_LdapConnection("localhost");  
L.bind();  
// Execute LDAP operations  
L.unbind();  
delete L;
```

### 7.14.6 bindSasl method

This method opens an LDAP session with SASL.

---

```
bindSasl(bind_rdn, mechanism, credentials, userid, realm)
```

---

#### Parameters

*bind\_rdn*

Specifies an entry in the LDAP directory

*mechanism*

Specifies the desired SASL mechanism

*credentials*

Specifies the credentials of the required SASL mechanism

*userid* Specifies a user ID.

*realm* Specifies a workgroup. If this is not required, enter an empty string.

#### Return value

None

#### Example

```
L=new WT_LdapConnection("localhost");
L.setOption("LDAP_OPT_PROTOCOL_VERSION","LDAP_VERSION3");
L.bindSasl("cn=admin,o=FTS,c=de", "DIGEST-MD5", "secret", "admin", "");
// Execute LDAP operations
L.unbind();
delete L;
```

### 7.14.7 compare method

This method compares an attribute value with an entry.

---

```
compare(dn, attribute, value)
```

---

#### Return value

Boolean return value: `true` if the two objects are equal, `false` if not

If an attribute consists of several values, the value `true` is returned as soon as the first match is found.

#### Parameters

*dn* Distinguished name (DN) used to address the entry in the directory

*attribute*  
Attribute with the entry addressed by *dn*

*value* Value of the attribute *attribute* with the entry addressed by *dn* in the directory

#### Example

```
dn="cn= Harry G. Miller, ou=board, o=FTS, c=DE";
att="password";
val="secret";
L=new WT_LdapConnection("localhost");
L.bind();
r=L.compare(dn, att, val);
if (r == true)
{ // Match }
else
{ // No match }
L.unbind();
delete L;
```



### 7.14.8 deleteEntry method

This method deletes the entry specified in the *dn* parameter from the LDAP directory.

---

`deleteEntry (dn)`

---

#### Return value

None

#### Parameters

*dn* Distinguished name (DN) used to address the entry to be deleted

#### Example

```
dn="cn= Harry G. Miller, ou=board, o=FTS, c=DE";
L=new WT_LdapConnection("localhost");
L.bind();
L.deleteEntry(dn);
L.unbind();
delete L;
```

### 7.14.9 equals method

This method compares the calling LDAP object with the object transferred in the *object* argument for equality of class, attributes, and values.

---

`equals(object)`

---

#### Return value

Boolean return value: `true` if the two objects are equal, `false` if not

#### Parameters

*object* LDAP object with which the calling object is to be compared

### 7.14.10 explodeDn method

This method splits the specified DN, which is supplied by the `getDn` method, into its individual components. The individual components correspond to the RDNs.

---

```
explodeDn(dn, with_attrib)
```

---

#### Return value

Array consisting of the components of the DN

#### Parameters

*dn* DN of the entry to be exploded

*with\_attrib*

Specifies whether or not RDNs are to be displayed with attribute names.

If so, the *with\_attrib* parameter must be set to `true`. If *with\_attrib* is set to `false`, only the attribute values will be returned.

#### Example

```
basedn="o=FTS, c=DE";
filter="((ou=board)(cn=Harry G. Miller))";
scope="LDAP_SCOPE_SUBTREE";
justthese=new Array("ou","cn","mail");
L=new WT_LdapConnection("localhost");
L.bind();
Count=L.search(basedn, filter, scope, justthese);
Id=L.firstEntry();
if((dn=L.getDn(Id))!=NULL)
{
    document.writeln(dn);
    // Output:
    // cn=Harry G. Miller, ou=board, o=FTS, c=DE
    A=L.explodeDn(dn, true);
    document.writeln(A);
    // Output:
    // [cn=Harry G. Miller,ou=board,o=FTS,c=DE]
    B=L.explodeDn(dn, false);
    document.writeln(B);
    // Output:
    // [Harry G. Miller,board,FTS,DE]
}
L.unbind();
delete L;
```

### 7.14.11 firstEntry method

This method returns the result entry ID of the first entry of a particular result.

You can use the `firstEntry` and `nextEntry` methods to read result entries sequentially. In this case, the value returned by the `firstEntry` method is transferred to the `nextEntry` method as an argument.

---

`firstEntry()`

---

#### Return value

Result entry ID of the first entry

#### Parameters

None

#### *Example*

```
basedn="o=FTS, c=DE";
filter="((ou=marketing)(cn=Harry*))";
scope="LDAP_SCOPE_SUBTREE";
justthese=new Array("ou","cn","mail");
L=new WT_LdapConnection("localhost");
L.bind();
Count=L.search(basedn, filter, scope, justthese);
for(Id=L.firstEntry();Id!=0;Id=L.nextEntry(Id))
{
    if((dn=L.getDn(Id))!=NULL)
        document.writeln("DN: "+dn);
}
L.unbind();
delete L;
```

### 7.14.12 getClassname method

This method returns a string specifying the name of the class to which the calling object belongs.

---

```
getClassName()
```

---

#### Return value

String specifying the class to which the calling object belongs, in this case WT\_LdapConnection

#### Parameters

None

### 7.14.13 getDn method

This method returns the Distinguished Name (DN) of a result entry.

---

```
getDn(result_entry_id)
```

---

#### Return value

DN of an entry

#### Parameters

*result\_entry\_id*

ID of the entry whose DN is to be returned.

*result\_entry\_id* has to be a valid value.

*Example*

See `firstEntry`

### 7.14.14 getEntries method

This method makes it easier to read several result entries (together with their attributes and values). It returns all the information available on the result of a query to LDAP ([search method](#), see [page 249](#)) in the form of a WTScript object.

---

```
getEntries()
```

---

#### Return value

Object of the `Array` class, which contains an object for all found entries.

Each array element is an object with the following attributes:

- `attr`: Object containing the attributes of the found entry
- `dn`: DN of the found entry
- `count`: Number of attributes

Binary values are shown in the form of Base64-coded strings.

#### *Example*

```
basedn="o=FTS, c=DE";
filter="((ou=marketing)(cn=Harry*))";
scope="LDAP_SCOPE_SUBTREE";
justthese=new Array("ou","cn","mail");
L=new WT_LdapConnection("localhost");
L.bind();
Count=L.search(basedn, filter, scope, justthese);
ResultObject=L.getEntries();
L.unbind();
delete L;
```

### 7.14.15 `getOption` method

This method returns the current value of a particular option.

---

```
getOption(option)
```

---

#### Return value

String containing the value of *option*

#### Parameters

*option* Option whose value is to be returned

The following values can be specified for *option*:

```
"LDAP_OPT_DEREF"  
"LDAP_OPT_SIZELIMIT"  
"LDAP_OPT_TIMELIMIT"  
"LDAP_OPT_PROTOCOL_VERSION"  
"LDAP_OPT_ERROR_NUMBER"  
"LDAP_OPT_REFERRALS"  
"LDAP_OPT_RESTART"  
"LDAP_OPT_ERROR_STRING"  
"LDAP_OPT_MATCHED_DN"  
"LDAP_OPT_HOST_NAME"
```

These values are described in the following document:

<http://www.openldap.org/devel/cvsweb.cgi/~checkout~/doc/drafts/draft-ietf-ldapext-ldap-c-api-xx.txt>

#### Example

```
L=new WT_LdapConnection("localhost");  
L.bind();  
sizelimit=L.getOption("LDAP_OPT_SIZELIMIT");  
L.unbind();  
delete L;
```

### 7.14.16 modify method

This method modifies an existing entry in the LDAP directory.

---

```
modify(dn, entry)
```

---

#### Return value

None

#### Parameters

*dn* Distinguished name (DN) used to address the entry to be modified

*entry* Information on the entry *dn*

#### Example

```
dn="cn=Harry G. Miller, ou=board, o=FTS, c=DE";
entry = new object();
entry.mail="Harry.G.Miller@my_company.net";
L=new WT_LdapConnection("localhost");
L.bind();
L.modify(dn, entry);
L.unbind();
delete L;
```

### 7.14.17 nextEntry method

This method returns the result entry ID of the next result entry.

You can use the `firstEntry` and `nextEntry` methods to read result entries sequentially. In this case, the value returned by the `firstEntry` method is transferred to the `nextEntry` method as an argument. Similarly, the value returned by each subsequent `nextEntry` call is in turn passed to the next `nextEntry` call as an argument.

---

```
nextEntry(result_entry_id)
```

---

#### Return value

Result entry ID of the next entry

#### Parameters

*result\_entry\_id*

Result whose entries are to be output

*Example*

See `firstEntry`



### 7.14.18 search method

This method searches the LDAP directory tree based on the defined filter. It returns information on a base entry (*base\_dn*) in the LDAP directory in accordance with the specified scope. Optional parameters are also provided for limiting the search result further.

---

```
search(base_dn, filter, scope)
search(base_dn, filter, scope, attributes)
search(base_dn, filter, scope, attributes, sizelimit)
```

---

#### Return value

Number of entries

#### Parameters

*base\_dn*

Base entry at which the search is to begin

*filter*

Filter for the search operation, which may contain boolean operators (for further information on filters, see the "Netscape Directory SDK").

The general syntax for a filter is as follows:

$$\textit{filter} ::= (\textit{attribute comp\_op value}) \mid (\textit{boolean\_op} (\textit{filter1}) [ (\textit{filter2}) ] \dots)$$

*Syntax elements*

*attribute*

An attribute

*comp\_op*

A comparison operator

The following comparison operators are available:

=

The system searches for all entries containing the attribute *attribute* with the value *value*.

*Example*

(cn=Harry)

Result: All entries in which cn=Harry

&gt;=

The system searches for all entries containing the attribute *attribute* with a value greater than or equal to *value*.

*Example*

(cn>=Harry)

**Result:** All entries in the range cn=Harry to cn=Z...

&lt;=

The search searches for all entries containing the attribute *attribute* with a value less than or equal to *value*.

*Example*

(cn<=Harry)

**Result:** All entries in the range cn=A... to cn=Harry

~=

The system searches for all entries containing the attribute *attribute* with a value approximately equal to *value*.

*Example*

(cn~=Meier)

**Result:** Entries like cn=Meier and cn=Meyer, for example

=\*

The system searches for all entries containing the attribute *attribute*.

*Example*

(cn=\*)

**Result:** All entries containing the attribute cn

*boolean\_op*

A boolean operator

The following boolean operators are available:

- & The system searches for all entries that fulfill the criteria of all specified filters.
- | The system searches for all entries that fulfill the criterion of at least one of the specified filters.
- ! The system searches for all entries that do **not** fulfill the specified filter criterion. This operator can only be applied to a single filter.

For example, the expression `!(filter1)` is permitted, but the expression `!(filter1)(filter2)` is not.

*Example*

```
(|(cn=Harry)(cn=Henry))
```

**Result:** All entries containing the attribute `cn` with the value `Harry` or `Henry`

*value*

Specifies a value of the attribute *attribute* to be used as a filter criterion

Wildcards are permitted here, which means that you can search for entries containing the attribute *attribute* with a value

- containing a specified character string
- beginning with a specified character string
- ending with a specified character string

*Example*

```
(cn=H*)
```

**Result:** All entries containing the attribute `cn` with a value beginning with the letter `H`

Further examples of filters can be found in the [section “WebTransactions and LDAP: examples” on page 255](#).

*scope*

Range of the search operation

Possible values:

- "LDAP\_SCOPE\_SUBTREE"  
Returns all information underneath the entry specified in *base\_dn*
- "LDAP\_SCOPE\_BASE"  
Returns all information relating to the entry specified in *base\_dn*
- "LDAP\_SCOPE\_ONELEVEL"  
Returns only information on the level directly underneath the entry specified in *base\_dn*

*attributes*

Array of attribute names (DN is always displayed) which allows you to restrict the attributes and values returned by the server:

e.g. `a=new Array("mail","sn","cn");`

*sizelimit*

Limits the number of entries found. If this is set to 0, then there is no limit.

*Example*

```
basedn="o=FTS, c=DE";
filter="((ou=marketing)(cn=Harry*))";
scope="LDAP_SCOPE_SUBTREE";
justthese=new Array("ou","cn","mail");
L=new WT_LdapConnection("localhost");
L.bind();
Count=L.search(basedn, filter, scope, justthese);
ResultObject=L.getEntries();
L.unbind();
delete L;
```

Further examples see [page 255](#).

### 7.14.19 setOption method

This method sets an option to a particular value.

---

```
setOption(option, newval)
```

---

#### Return value

None

#### Parameters

*option* Option to be set

Using the *newval* parameter, you can specify values for the following options:

```
"LDAP_OPT_DEREF"  
"LDAP_OPT_SIZELIMIT"  
"LDAP_OPTTIMELIMIT"  
"LDAP_OPT_PROTOCOL_VERSION"  
"LDAP_OPT_REFERRALS"  
"LDAP_OPT_RESTART"  
"LDAP_OPT_HOSTNAME"  
"LDAP_OPT_VERSION3"
```

*newval* Value to which the option is to be set

#### Example

```
L=new WT_LdapConnection("localhost");  
L.bind();  
L.setOption("LDAP_OPT_SIZELIMIT", "100");  
L.unbind();  
delete L;
```

### 7.14.20 toString method

This method returns a string listing each attribute and the associated value. If an attribute is an object of type `function`, then the function definition is returned instead of the value.

You can use this method to create a new object with identical attribute values.

In order to avoid endless chains, the `toString` method will terminate output in the event of recursion, i.e. output will be stopped as soon as the same object reference is encountered a second time.

---

`toString()`

---

#### Return value

String listing each attribute and the associated value

#### Parameters

None

### 7.14.21 unbind method

This method closes an LDAP session.

---

`unbind()`

---

#### Return value

None

#### Parameters

None

*Example*

See `bind` and `bindSasl`

### 7.14.22 valueOf method

This method returns a reference to the calling object.

---

```
valueOf()
```

---

#### Return value

Reference to the calling object

#### Parameters

None

### 7.14.23 WebTransactions and LDAP: examples

This section contains three examples of how to use the `WT_LdapConnection` class:

- a search operation
- a comparison operation
- LDAP exception handling in `WTScript`

#### Example of a search operation

In this example, a search is carried out on the "ou" (organizational unit), "sn" (surname), "givenname", and "mail" (email address) attributes for all those in the organization "FTS" whose surname begins with the letter "H" and whose first name is "John".

```
<wtOnCreateScript>
dn = "o=FTS, c=DE";
filter = "(|(sn=H*)(givenname=John))";
justthese = new Array( "ou", "sn", "givenname", "mail");
scope = "LDAP_SCOPE_SUBTREE";
l = new WT_LdapConnection("localhost");
l.bind();
// Search the directory
count = l.search(dn, filter, scope, justthese);
info = l.getEntries();
document.writeln(info);
l.unbind();
delete l;
</wtOnCreateScript>
```

### Example of a comparison operation

In this example, the specified password is compared with the password entry in the directory specified by the DN.

```
<wtOnCreateScript>
dn = "cn=Suzy Meier, ou=development, o=FTS, c=DE";
value = "secretpassword";
attr = "password";
l = new WT_LdapConnection ();
l.bind();
// Compare the value of the specified attribute with the directory entry
if (l.compare(dn, attr, value)) {
    document.writeln("Password correct.");
}
else {
    document.writeln("Password invalid. Please try again.");
}
l.unbind();
delete l;
</wtOnCreateScript>
```



### Example of exception handling in WTSript for the exception `LdapError`

In this example, the `try` block guards the sequence of statements between (1) and (2) before the `LdapError` exception occurs. The subsequent `catch` block then executes the error handling procedure. Further information on exception handling in WTSript can be found in the [section "Exception handling procedure" on page 305](#).

```
<wtOnCreateScript>
dn = "o=FTS, c=DE";
filter = "(|(sn=H*)(givenname=John))";
justthese = new Array( "ou", "sn", "givenname", "mail");
scope = "LDAP_SCOPE_SUBTREE";
try
{
    ----- (1)
    l = new WT_LdapConnection("localhost");
    l.bind();
    // Search the directory
    count = l.search(dn, filter, scope, justthese);
    info = l.getEntries();"LDAP_OPT_
    document.writeln(info);
    l.unbind();
    delete l;
}
----- (2)
catch(LdapError)
{
    document.writeln(LdapError.text);
    switch(LdapError.msg#)
    {
        case 303:
            l.unbind();
            break;
        default:
    }
    delete l;
}
</wtOnCreateScript>
```

## 7.15 WT\_Userexit class

Objects of the `WT_Userexit` class allow you to call user exits. Each object of this class corresponds to one user exit library. The functions defined in this library are then available to you as methods of this object.

### 7.15.1 Constructors

---

```
WT_Userexit()  
WT_Userexit(library)
```

---

The *library* specification allows you to enter the name of a library. User exits which are called via this object are searched for in the specified library.

The name is specified without a suffix. WebTransactions adds the suffix `.so` or `.dll` depending on the operating system. It looks for the specified library in the base directory and - if no corresponding library is found there - in the installation directory.

If *library* is not specified, the default library `WTUserexits.[so|dll]` is used.

## 7.15.2 Methods

All user functions present in shared libraries or integrated in WebTransactions are available as methods of `WT_Userexit` class objects. A *function* user exit can thus be called in the script area and within an evaluation operator:

---

```
mylib.function(...);  
WT_Userexit.function(...);
```

---

*mylib* is an object of the class `WT_Userexit`. The function *function* is searched for in the library on which the object is based.

You can use the notation `WT_Userexit.function` to access the class methods of the `WT_Userexit` class: *function* is then looked for in the library `WTUserexits.[d11|so]` (initially in the base directory and - if it is not located there - in the installation directory).



Dynamic libraries are not supported under BS2000/OSD. In this case, you must link the functions to the WTHolder program.

### *Example*

In the following example, the user function `myFunction` is called in the library `myLibrary.so` which is located in a user-defined subdirectory named `myDir`.

```
myUserExit = new WT_Userexit( 'myDir/myLibrary' );  
res = myUserExit.myFunction(1,2,3);
```



---

## 8 WTML tags

WTML tags contain the WebTransactions-specific functions. They have the form `<wt...>` or `</wt...>` and therefore correspond to the HTML standard.

WTML tags are interpreted by WebTransactions when the template is read. The process is as follows:

- evaluate the control structures and statements which control construction of the HTML page
- execute the actions and statements which prepare the data for construction of the HTML page
- buffer actions and statements for follow-up processing of the response data received from the browser

Any characters located outside the surrounding angle brackets belong not to the WTML tag but to HTML or script areas. If they belong to HTML areas, they are normally displayed by the browser - with the following two exceptions:

- If there are only blanks and tabs between the start of the line and the opening angle bracket, then these belong to the WTML tag and are ignored during output.
- If a WTML tag ends at the end of a line, the following line feed belongs to the WTML tag and is ignored on output.

This approach makes it possible to write WTML tags clearly in separate lines and structure them by means of indents without impairing the appearance of the template.

## Overview of WTML tags

The WTML tags available are listed in the two tables below. These are followed by detailed descriptions of the individual WTML tags. For examples of ways in which the WTML tags interact, refer to chapter 11.

WTML tag	Function	Syntax
Rem tag	Inline documentation of the template; ignored on generation of the HTML page.	<wtRem ...> or <wtRem> ... </wtRem>
Dataform tag	Marks the area for the data resent to the WebTransactions application; corresponds to the HTML Form tag.	<wtDataform ...> ... </wtDataform ...>
Exit tag	Terminates processing of the current area.	<wtExit ...>
Include tag	Points to other included template files, e.g. layouts for a uniform, easily modifiable Look & Feel for HTML pages.	<wtInclude ...>
OnCreateScript tag	Opening and closing OnCreateScript tags delimit a WTScrip script which is executed directly when the template is read (for notes on use, see OnCreateScript tag).	<wtOnCreateScript> ... </wtOnCreateScript>
OnReceiveScript tag	Opening and closing OnReceiveScript tags delimit a WTScrip script which is executed when data is received from the browser (for notes on use, see OnReceiveScript tag).	<wtOnReceiveScript> ... </wtOnReceiveScript>

The OnCreateScript/OnReceiveScript tags can be differentiated by their execution time:

- OnCreateScript tags are executed immediately the template is read, and the value assignments take effect directly, i.e. on generation of the current HTML page.
- OnReceiveScript tags are not executed until the HTML page has been generated and sent to the browser and the data sent by the browser has been received, i.e. before the next template is read. They are suitable for the follow-up processing of data from the current dialog.

The following WTML tags are available for control structures:

WTML tag	Function	Syntax
If/Else/Endif tag	Simple selection on the basis of the condition.	<wtIf ...> ... <wtElse> ... <wtEndIf>
DoWhile tag	Loop which is repeated until the continuation condition becomes invalid.	<wtDoWhile ...> ... </wtDoWhile>
Do/Until tag	Loop which is repeated until the exit loop condition becomes valid.	<wtDo> ... <wtUntil(...)>

### Keywords in WTML tags

The WTML tags and the keywords which occur in them can be written in any combination of uppercase and lowercase characters. Like HTML tags, WTML tags contain a sequence of properties which are designated by certain keywords and may occur in any order. The keyword is followed by an equals sign and a value for this property. The value can be specified as a string or a simple string expression.

### Omission of quotes

If the string which represents a value within a WTML tag contains no blanks, tabs, or line feeds (or if any such characters are invalidated), then the enclosing quotes may be omitted: if the first character of the value is not a quote, then all the characters up to the first white space or up to the terminating bracket (“>”) are considered to be values.

#### *Example*

```
<wtInclude Name=hello\ world>
```

## 8.1 Rem - inserting comments

This tag allows you to insert inline documentation comments in the template. In contrast to HTML comments, these comments are not sent to the browser. `Rem` tags may be located in the HTML area as well as within WTML tags, and may be positioned anywhere where white spaces are permitted. They are not recognized within string expressions (i.e. evaluated as a fixed string).

---

```
<wtRem comments>
```

---

```
<wtRem>  
  comments  
</wtRem>
```

---

Two alternative formats are possible for `Rem` comments:

- In the first format, the comment is located directly inside the pointed tag brackets. In this variant, the comment may not contain the “>” character since this character would be interpreted as the end of the comment. Nested or empty comments are not permitted in this format.
- In the second format, the comment is located between an opening and a closing `Rem` tag. In this case, the comment may contain the “>” character. This makes it possible to nest comments as you wish, or deactivate WTML tags with comments. Empty comments are also permitted.



## 8.2 Dataform - defining form areas

Opening and concluding `Dataform` tags enclose a form area. Within this form area, you use the usual HTML tools to define dialog elements such as buttons, text boxes, multiline text boxes (text areas), or selection lists. The user working with the browser can then enter data and select functions. When the user presses the `Submit` button, the entries are sent to `WebTransactions`.

The `WebTransactions Dataform` tag has the following form:

---

```
<wtDataform [Name="name"] [OnSubmit="OnSubmitHandler"] [ASYNC_PAGE="asyncPage"]>
area
</wtDataform>
```

---

*name* Any name

*OnSubmitHandler*

JavaScript code which is executed by the browser once the user has pressed the `Submit` button (even before the data is sent). The browser may, for example, subject the user input to a plausibility check.

*asyncPage*

If you set this attribute, then `WebTransactions` processes the page even if it does not fit into the sequence of dialog steps (asynchronous communication). This means that you can send `WebTransactions` a page which does not correspond to the last page to be output. *asyncPage* specifies the template which processes this asynchronous request (for more information, see the `WebTransactions` manual "Concepts and Functions").

*area* Here you use the usual HTML tools to define dialog elements such as buttons or text boxes. The area may also contain HTML text.

At runtime `WebTransactions` replaces the `Dataform` tags with "normal" HTML form tags:

```
<Form Method=POST Name="name" ONSUBMIT="OnSubmitHandler"
Action=url_of_webtransactions>
<Input Type="HIDDEN" ...> ...
Area
</Form>
```

The `Action` attribute is automatically set to the value of the URL of the `WebTransactions` CGI component `WTPublish`, to which the completed form is to be sent. In addition, `WebTransactions` generates a series of hidden fields which are also sent. These hidden fields help, for example, to relocate the session.

Currently, WebTransactions simply converts the concluding `</wtDataform>` tag into a `</Form>`. However, in order to take advantage of extensions to the concluding tag in later versions, it is recommended that you use the WTML tag rather than the HTML form tag to terminate the area to be sent.

There may also be multiple areas of the form `<wtDataform> ... </wtDataform>` within a template. In this case, when a WebTransactions application's `Submit` button is pressed, only the corresponding form area and associated data are sent to it.

### Alternative notation

Some browsers (e.g. Netscape) do not display dialog elements which are not surrounded by `<Form> ... </Form>`. If you use this type of browser when programming your templates but nevertheless want to check the template layout off-line in the browser, you can use the following notation for the `Dataform` tag:

---

```
<Form WEBTRANSACTIONS [Name="name"] [OnSubmit="submitFunction"]>  
Area  
</Form>
```

---

This notation has the disadvantage that the tag does not start with `<wt>` and is therefore not differentiated from the standard HTML tags.

## 8.3 Exit - terminating processing

This tag allows you to terminate processing in the specified area of the current template. It may be located anywhere within the HTML area. Any `Receive` rules that have so far been identified are executed.

---

```
<wtExit scope={"TEMPLATE"|"DIALOGSTEP"|"SESSION"}>
```

---

`scope` Defines where processing is to be terminated and where it is to be recommenced:

`TEMPLATE`

Processing of the current template is terminated. It then continues with the next statement in the calling template. If called at the topmost level, `scope="TEMPLATE"` has the same effect as `scope="DIALOGSTEP"`

`DIALOGSTEP`

Processing of all the templates involved in this dialog step is terminated.

`SESSION`

The current `WebTransactions` session is terminated at the next possible opportunity. The result of the current WTML document is the last page sent to the browser.

If this parameter is set, then the behavior is equivalent to `WT_SYSTEM.EXIT_SESSION="TRUE"`.

### See also

[“exitDialogStep\(\) function” on page 90](#), [“exitSession\(\) function” on page 94](#), and [“exitTemplate\(\) function” on page 95](#)

## 8.4 Include - including templates

This tag allows you to include a template. It may be located anywhere in the HTML area.

The included template is fully inserted. Within the included template, you can use the same language components as in any other template. However, in any template the WTML tags must be syntactically complete, e.g. it is *not* permissible to start an IF control structure in the higher-level template and then finish it in the included template.

---

```
<wtInclude Name="fileName">
```

---

### *fileName*

Name of the template which is to be included. *fileName* is a relative file name. You do not need to specify the file name's `.htm` suffix. WebTransactions searches for the corresponding template on the basis of the set language and style.

You can specify *fileName* as a fixed string or as a simple string expression.

### *Example*

```
<wtInclude Name="header">
```

You can, for example, include the definition of a page header to give your HTML pages a uniform Look & Feel which can easily be modified later. The `header` template can contain both HTML areas and WTML tags (for the modification of the data it displays).

## 8.5 IF/ELSE/ENDIF control structure

The IF control structure may be located anywhere within the HTML area. It can contain constant HTML text, standard HTML tags, and WTML tags. You can nest IF control structures as you wish.

---

```
<wtIf (Condition)>
  Block1
[<wtElse>
  Block2]
{<wtEndIf> | </wtIf>}
```

---

### *Condition*

Any expression. For reasons of compatibility, the numerical comparison operators (`# ==`, `# !=`, `# >`, `# <`, `# <=`, `# >=`) can also be used in these expressions, see also section [“Comparison operators which force a numerical comparison \(only in WTML tags\)” on page 67](#). WebTransactions evaluates *Condition* when it reads the template, converts the expression to type `boolean` if necessary and, depending on the result, interprets either *Block1* or *Block2*: If the result true, *Block1* is executed. Otherwise *Block2* is executed— if available.

### *Block1*

### *Block2*

Depending on the evaluation of the condition when the template is read, either *Block1* or *Block2* is processed. These blocks may contain constant HTML text, standard HTML tags, or WTML tags which may or may not be processed depending on the condition. *Block2* (the ELSE branch) is optional.

*Example*

You want to add a button to the generated formats, which allows the user to actively close the session.

```
<wtIf ( "##WT_POSTED.ExitButton#" != "cancel")>
<wtREM HTML generation in accordance with automatically converted templates
    including all OnReceive tags!>
    <wtDataform Name="exitForm">
        You can close the session if you wish
        <Input Action="SUBMIT" Name="ExitButton" Value="cancel">
    </wtDataform>
<wtElse>
    <wtoncreatescript>
    <!--
        WT_HOST.OSD_0.close();
        exitSession();
    //-->
    </wtoncreatescript>
End, You have terminated the session, Thank you for visiting us.
<wtEndIf>
```



WebTransactions interprets the IF constructor when reading the template. The condition is therefore always **checked during the interpretation of the template**. Even if you write `OnReceiveScript` tags in a branch of the IF control structure, the condition is evaluated **at the moment the HTML page is generated**. Depending on whether or not the condition is valid, the actions in the `OnReceiveScript` tag are either buffered for subsequent execution or ignored.

This type of IF control structure is therefore **not** suitable if you want to make the execution of processing steps dependent on conditions which are not evaluated until `Receive` time. For example, it is not possible to write a branch which is dependent on the posted data. In such cases, you should use the `WTScript if` control structure (within an `OnReceiveScript` tag). The `WTScript if` control structure is described on [page 279](#).

## 8.6 DO WHILE loop

DO WHILE loops can be used anywhere within the HTML area. When reading the HTML page, WebTransactions continues to evaluate the template area *Block* as long as *Condition* is valid. Processing of the template then continues after the loop construct.

---

```
<wtDoWhile (Condition)>  
Block  
</wtDoWhile>
```

---

### *Condition*

See description of the IF control structure ([page 269](#))

*Block* The template area *Block* may contain standard HTML text, standard HTML tags, or WTML tags.



DO WHILE loops are always processed **at the moment the HTML page is generated**. The condition is therefore always checked during the interpretation of the template. If the body of the loop contains `OnReceiveScript` tags, then the corresponding processing steps are buffered for subsequent execution.

DO WHILE loops of this type are therefore **not** suitable if you want to make the number of iterations dependent on conditions which are not evaluated until `Receive` time (e.g. dependent on posted data). In such cases use the `WTScrip` `while` or `for` loops (within an `OnReceiveScript` tag). The `WTScrip` loops are described starting on [page 281](#).

## 8.7 DO UNTIL loop

DO UNTIL loops can be used anywhere within the HTML area. When interpreting the HTML page, WebTransactions continues to evaluate the template area *Block* as long as *Condition* is valid. Processing of the template then continues after the loop construct.

---

```
<wtDo>
Block
<wtUntil(Condition)>
```

---

### *Condition*

See description of the IF control structure ([page 269](#))

*Block* The template area *Block* may contain standard HTML text, standard HTML tags, or WTML tags.

### *Example*

Your host application contains a format which can be scrolled through page by page. It contains a box named POSITION in which the following entries can be made: + (page down), - (page up), ++ (to end), and -- (to start). As long as the current page is not the last page, the box contains the entry "+". When the end of the list is reached, the host application enters the value "-" in the box. You can program the following loop to display the entire list on an HTML page:

```
<wtRem Position at start of list>

<wtoncreatescript>
<!--
    WT_HOST.OSD_0.POSITION.Value="--";
-->
</wtoncreatescript>

<wtDo >
    <wtRem Read one page from host application>
    <wtoncreatescript>
    <!--
        WT_HOST.OSD_0.send();
        WT_HOST.OSD_0.receive();
    -->
    </wtoncreatescript>

    <wtRem Output data on HTML page>
    ##ListLine1.Value# <br>
    ##ListLine2.Value# <br>
    ##ListLine3.Value# <br>

    <wtRem Read next page if end not yet reached>
```



```
<wtIf ("##POSITION.Value#" !="--")>
  <wtoncreatescript>
  <!--
    WT_HOST.OSD_0.POSITION.Value="+";
  //-->
  </wtoncreatescript>
<wtEndIf>
<wtUntil("##POSITION.Value#" =="--")>
```



DO UNTIL loops are always processed **at the moment the HTML page is generated**. The condition is therefore always checked during the interpretation of the template. If the body of the loop contains `OnReceiveScript` tags, then the corresponding processing steps are buffered for subsequent execution.

DO UNTIL loops of this type are therefore **not** suitable if you want to make the number of iterations dependent on conditions which are not evaluated until `Receive` time (e.g. dependent on posted data). In such cases, use the `WTScrip` `while` or `for` loops (within an `OnReceiveScript` tag). The `WTScrip` loops are described starting on [section “while loop” on page 281](#).

## 8.8 OnCreateScript - WTScrip at generation time

Between the opening and closing `OnCreateScript` tags, you can formulate a program in WTScrip. This is run by WebTransactions during the interpretation of the template and thus **before** the page is sent to the browser. The script is therefore executed on the WebTransactions server (“server-side” scripts).

This differentiates WTScrip programs within `OnCreateScript` tags from JavaScript programs within “normal” HTML `<script>` tags which are interpreted by the browser (“client-side” JavaScript).

---

```
<wtOnCreateScript>  
CreateScript  
</wtOnCreateScript>
```

---

### *CreateScript*

WTScrip program which is executed by WebTransactions during the interpretation of the template. The WTScrip statements which you can use here are described in [chapter “WTScrip statements \(in OnCreateScript/OnReceiveScript\)” on page 277ff.](#)

If you first want to view the template off-line in the browser, you can improve the display by hiding the WTScrip program in an HTML comment:

```
<wtOnCreateScript>  
<!--  
CreateScript  
// -->  
</wtOnCreateScript>
```

## 8.9 OnReceiveScript - WScript after the receipt of browser data

Between the opening and closing `OnReceiveScript` tags, you can formulate a program in WScript. WebTransactions inserts this in the sequence of the other `OnReceive` tags. Its execution is therefore postponed until the data sent by the browser has been received, i.e. until the current HTML page has been generated and sent to the browser and the data sent by the browser has been received.

Like WScript programs within `OnCreateScript` tags, WScript programs within `OnReceive` tags are interpreted by WebTransactions and not by the browser (“server-side” scripts).

---

```
<wtOnReceiveScript>  
ReceiveScript  
</wtOnReceiveScript>
```

---

### *ReceiveScript*

Program in WScript which is executed by WebTransactions after the receipt of data posted by the browser. The WScript statements which you can use here are described in [chapter “WScript statements \(in OnCreateScript/OnReceiveScript\)” on page 277ff.](#)

If you first want to view the template off-line in the browser, you can improve the display by hiding the WScript program in an HTML comment:

```
<wtOnReceiveScript>  
<!--  
ReceiveScript  
// -->  
</wtOnReceiveScript>
```



---

## 9 WTScrip statements (in OnCreateScript/OnReceiveScript)

Within the WTScrip areas which you introduce with `<wtOnCreateScript>` or `<wtOnReceiveScript>`, you can specify WTScrip statements which are executed at the server. WTScrip statements are based on JavaScript V1.2 and the same language concepts are therefore generally supported. The only difference is that the JavaScript object model has been replaced by a separate model, which is described in the WebTransactions manual “Concepts and Functions”. This section describes the server-side WTScrip statements supported by WebTransactions.

Browsers output unknown WTScrip code as clear text. You can prevent them from doing this by enclosing the scripts in comments: `<!--` at the start and `//-->` at the end.

### Overview of statements

WTScrip statements can contain expressions and other statements, and must be terminated with a semi-colon (;). They are executed in sequence. The statements may have side effects, such as the evaluation of expressions or the assignment of values to variables. However, unlike expressions, the statements themselves do not have any value or data type.

This chapter describes the action of the individual statements during the processing of a WTScrip script. WebTransactions supports the following statements:

- Empty statements
- Statements for sequence control (conditional branches and loops):  
`if`, `while`, `do/while`, `for`, `for/in`, `switch`, `break`, `continue`, `return`
- Statements for the declaration of variables and functions: `var` and `function`
- `with` expressions and statements which provide a shorter way of writing object references in statements
- Statements for troubleshooting: `throw`, `try`, `catch`, `finally`

## 9.1 Empty statements

No action is executed for an empty statement.

---

```
;
```

---

*Example*

The following loop searches the array `a` for the first undefined element:

```
for(i=0; i<a.length && a[i]; i++);
```

## 9.2 Expression as a statement

An expression can be made into a statement by terminating it with a semi-colon. Such expressions include, for example, assignments, function calls, and the increment/decrement operators.

---

*expression* ;

---

*expression*

Any expression (see [chapter “Expressions and operators” on page 63](#))

### Description

The *expression* expression is evaluated.

*Examples*

```
output = "Hello" + name;
WT_SYSTEM.STYLE = WT_POSTED.STYLE;
    //Value assignment
document.write("welcome, " + name);
    //Call to document object's write method
counter++; //Increment operator
6*7; // Statement of no significance
```

## 9.3 Statement block as a statement

A statement block may consist of no, one, or more statements which are enclosed in braces.

---

```
{  
  [statement] . . .  
}
```

---

### *statement*

Individual statement or statement block. These may include, for example, assignments, function calls, and the increment/decrement operators.

### **Description**

A statement block is executed by executing the individual statements in order.

## 9.4 Sequence control statements

Statements used for sequence control purposes, also known as control structures, control the sequence in which a program is processed. You use conditions to specify which statements are executed (branching) and how often they are executed (loops).

### 9.4.1 if branch

The `if` branch executes a statement if the specified condition is fulfilled. If the condition is not fulfilled (false), then other statements may (optionally) be executed. If multiple statements are to be made dependent on the condition, then these must be enclosed in braces.

The condition may be any expression which can be mapped to a logical value (`true/false`) (see [chapter “Expressions and operators” on page 63](#)). The statement may be nested.

---

```
if ( condition ) block1 [ else block2 ]
```

---

### *condition*

Expression which represents the condition and which is evaluated

### *block1, block2*

A statement or a statement block

## Description

The expression *condition* is evaluated and converted to data type `boolean`. If the result is the value `true`, then *block1* is executed; if it is the value `false`, then *block2* is executed.

If nested `if` structures are used, the `else` branch always belongs to the innermost `if` statement. No other `else` branch may be associated with this `if` statement.

```
if ( expression1 )
  if ( expression2 )
    block1
  else
    block2
```

The first `if` statement contains only a `true` branch which consists of the second `if` statement.

### Example 1

You can use an `if` branch, for example, to create a variable as a reference to the global or connection-specific system object in the start template, provided such an object exists. If a connection-specific system object (which is not undefined) exists for the connection, then the `host_system` variable is set to the value of the connection-specific system object. Otherwise, it is set to that of the global system object.

```
if (WT_HOST.myComObj.WT_SYSTEM != null)
  host_system = WT_HOST.myComObj.WT_SYSTEM;
else
  host_system = WT_SYSTEM;
```

### Example 2

Communication with the host application is only performed if the style has not changed.

```
if (WT_SYSTEM.TravStyle == "NoChange")
{
  host.send();
  host.receive();
}
```



## 9.4.2 while loop

You can use a `while` loop to execute and repeat statements as long as the loop condition is fulfilled (true). If the condition is not fulfilled (false), loop processing is terminated and then continued using the statement directly after the `while` loop.

---

```
[label:] while ( condition ) block
```

---

*label* Label which allows you to refer to a `break` or `continue` statement

*condition*

Expression which is checked before the `while` loop is executed for the first time and before every repetition

*block* A statement or a statement block

### Description

The expression *condition* is evaluated repeatedly and converted to data type `boolean`. *block* continues to be executed as long as the result of the evaluation is the value `true`.

If the loop condition is always met, then an infinite loop occurs.

If *block* contains a `break` statement, this has the following effect:

- `break` without *label*

The loop is aborted and processing continues with the statement directly after the loop.

- `break` with *label*

The nearest statement identified in *label* is aborted and processing continues with the statement immediately after the aborted statement.

If *block* contains a `continue` statement, this has the following effect:

- `continue` without *label*

The statement or statement block specified in *block* is aborted. *condition* is reevaluated and processing of the `while` loop continues accordingly.

- `continue` with *label*

The statement or statement block specified in *block* is aborted. The value specified in the *condition* parameter of the loop identified in *label* is reevaluated and processing continues with the loop identified in *label*.

*Example of an infinite loop*

```
while(true)
{
}
```

*Example*

The following `while` loop outputs the defined values of array `a` in the columns of a table row:

```
document.write("<tr>");
i=0;
while(i < a.length && a[i])
{
    document.write("<td>" + a[i++] + "</td>");
}
document.write("</tr>");
```

### 9.4.3 do/while loop

You can use `do/while` loops to execute and repeat statements for as long as the condition is fulfilled (true). If the condition is not fulfilled (false) loop processing is terminated and then continued with the statement directly after the `do/while` loop.

The statements are run in all cases at least once before the condition is checked.

---

```
[label:] do block while ( condition );
```

---

*label* Specifies a label which allows you to refer to from a `break` or `continue` statement.

*block* A statement or a statement block.

*condition*

An expression that is checked after each loop run. The expression is converted to data type `boolean` and as long as this returns `true`, the loop is continued.

#### Description

If *block* contains a `break` or `continue` statement, the execution flow is as described in [section “while loop” on page 281](#).

#### Example

```
a = 0;
arr = new Array;
do {
    arr[a] = a;
}
while ( a++<100 );
```

### 9.4.4 for loop

The `for` loop allows you to repeat the execution of a statement or statement block for as long as the loop condition is fulfilled.

---

```
[label:] for ( [init ]; [condition ]; [update ] ) block
```

---

*label* Label which allows you to refer to a `break` or `continue` statement

*init* An expression or a variable declaration. This corresponds to a declaration made prior to the `for` loop.  
*init* is executed once at the start of the loop. Typically, it is used to initialize a loop counter.

The following steps are executed repeatedly:

*condition*

*condition* is evaluated and converted to the data type `boolean`. If the result is `true` or *condition* is not available, *block* is executed. If the result is `false`, then the loop is terminated.

*update* Expression which is evaluated once the body of the loop has been processed. This typically modifies the value of the loop counter.

*block* Statement or statement block which forms the body of the loop

#### Description

If *block* contains a `break` statement, this has the following effect:

- `break` without *label*

The loop is aborted and processing continues with the statement directly after the loop.

- `break` with *label*

The nearest statement identified in *label* is aborted and processing continues with the statement immediately after the aborted statement.

If *block* contains a `continue` statement, this has the following effect:

- `continue` without *label*

The statement or statement block specified in *block* is aborted and *update* executed. The value specified in the *condition* parameter is reevaluated and processing of the loop continues accordingly.

- continue with *label*

The statement or statement block specified in *block* is aborted. The value specified in the *condition* parameter of the loop identified by *label* is reevaluated after *update* is executed , and processing continues with the loop identified in *label*.

### Example 1

In the following example, a counter *i* is defined and initialized with the value 1. Each time the loop is repeated, the value of the counter is increased by 1. When its value is greater than 100, the condition of the second statement is not fulfilled any longer and the loop is terminated.

The loop counter is used to form the square of the current value on each loop run. The result is output in the current HTML document (`document.write` method, see [section “write / writeln method” on page 153](#)).

```
for(i = 1; i <= 100; i++)
{
    var x = i * i;
    document.write("<br>The square of " + i + " is " + x);
}
```

### Example 2

The following `for` loop allows you to read all the fields of the current screen (if there is no further field present, `$END` is returned). You can use the `if` branch to check whether the end user has modified any values and assign these to the corresponding host objects of the OSD/MVS communication object.

```
...
for (element = WT_HOST.con.$FIRST.Name;
    element != "$END";
    element = WT_HOST.con.$NEXT.Name)
if ( WT_POSTED[element] != WT_HOST.con[element].Value )
    WT_HOST.con[element].value = WT_POSTED[element];
...

```

For examples of `for` statements with labels, which are target of a `break label` statement or and `continue label` statement, see the [section “break statement” on page 290](#) or the [section “continue statement” on page 292](#).

## 9.4.5 for/in loop

This statement loops through the attributes of an object (*object*). For example, these can be the name/value pairs, which the browser receives as the attributes of the posted object, or the elements of an array. The variable *name* stores the current attribute of *object*.

---

```
[label:] for ( [var] name in object ) block
```

---

*label* Label which allows you to refer to a `break` or `continue` statement

*name* Variable to which the name of an object attribute or the name/index of an array element (i.e. the property of an object) is assigned (see [section “Object class” on page 173](#)). The variable *name* is declared implicitly. If you call the `for` loop in a function and want to create the variable locally, you must use `var name` (see [section “var statement” on page 295](#)).

*object* Expression which defines the object whose attributes are looped through

*block* A statement or sequence of statements to be executed for each property

### Description

The expression *object* is evaluated and converted to data type `object` if necessary. The names of the properties are assigned to the variable *name* in string form. *block* is executed after each value assignment, for example in order to output the names of the object properties.

If *block* contains a `break` statement, this has the following effect:

- `break` without *label*

The loop is aborted and processing continues with the statement directly after the loop.

- `break` with *label*

The nearest statement identified in *label* is aborted and processing continues with the statement immediately after the aborted statement.

If *block* contains a `continue` statement, the statement or statement block specified in *block* is aborted and processing continues as follows:

- `continue` without *label*

The next attribute of the object is assigned to the variable and processing of the `for/in` loop continues accordingly.

- `continue` with *label*

The system jumps to the start of the loop identified in *label* where the next value is assigned to the loop variable. Processing of this loop is then continued.

*Example 1*

In this example, the function collects all the sales values (turnover) for the 1st quarter and outputs them as an HTML-formatted string.

```
...
function properties(turnover) {
    for (var i in turnover) {
        document.write(turnover[i] + "<p>")
    }
}
...
```

*Example 2*

This `for/in` loop outputs the names of all the host objects together with the property `HTMLValue`. The communication object for the connection is named "Conn".

```
for (obj in WT_HOST.Conn)
{
    document.write (obj + ":" + obj.HTMLValue + "<br>");
}
```

*Example 3*

If an HTML page contains a list of names from which the user can select multiple entries, then `WebTransactions` returns the selected entries in an array. If only one entry is selected, a string is returned. The following `for` loop outputs all the selected names. If only one entry is selected, then `multi` is not an array and the loop is not executed.

```
b = "Names:";
if (WT_POSTED.multi)
{
    for ( a in WT_POSTED.multi )
        b += " " + WT_POSTED.multi[a];
    if ( b == "Names:" )
        b += " " + WT_POSTED.multi;
}
else
{
    b+ = "Nothing selected!";
}
```

## 9.4.6 switch statement

This statement is used to select between multiple cases (*case*). One or more statements are executed if the value of an expression which is checked as the input condition corresponds to the value of the expression of a case (*label*). The `break` statement can be used to abort the sequential processing of the individual cases.

---

```
switch (expression1)
{
    case expression2: statement1 ... [break [label:];]
    case expression2: statement1 ... [break [label:];]
    ...
    [default:] statement2 ...
}
```

---

*expression1*

Any expression (see [chapter “Expressions and operators” on page 63](#))

*expression2*

Expression whose value is compared with the value of *expression1*.

*statement1*

One or more statements which are executed if the expressions *expression1* and *expression2* are equivalent.

*statement2*

One or more statements which are executed if non of the values in *expression2* are equivalent to *expression1*

`break [label]`

Aborts the `switch` statement and continues processing with the next statement. Processing continues with the statement immediately after the `switch` statement.

`break label` aborts the nearest statement containing the `switch` statement identified in *label*, and continues processing with the statement immediately after the aborted statement.

### Description

The expression *expression1* is evaluated and compared for lexical or numerical equivalence etc. with the *expression2* expressions which follow each `case` keyword. Execution of the script continues with the case for which the value of *expression2* corresponds to the value of *expression1*. This case and all the cases which follow it are processed.

If no identical expression is found, then any statements specified after the `default` keyword are executed. If `default` is not specified, then no actions are performed in the `switch` statement.



*Example 1*

You can use the following `switch` statement to map user input values, for example in a drop-down list, to host objects. `host` points to the host data objects for the connection.

```
switch (WT_POSTED.COUNTRY)
{
    case "Belgium": host.Country.Value= "1"; break;
    case "France": host.Country.Value= "2"; break;
    case "Germany": host.Country.Value= "3"; break;
    case "Greece": host.Country.Value= "4"; break;
    default: host.Country.Value= "0";
}
```

*Example 2*

A value is assigned to `WT_SYSTEM.STYLE` depending on the value posted in `FORMSTYLE`.

```
WT_SYSTEM.TravStyle = "change";
switch (WT_POSTED.FORMSTYLE)
{
    case "Forms": WT_SYSTEM.STYLE = "forms";
        break;
    case "Simple": WT_SYSTEM.STYLE = "simple";
        break;
    case "Green": WT_SYSTEM.STYLE = "green";
        break;
    case "Enhanced": WT_SYSTEM.STYLE = "enhanced";
        break;
    default: WT_SYSTEM.TravStyle = "NoChange";
        //WT_SYSTEM.STYLE remains unchanged!
        break;
}
```

## 9.4.7 break statement

This statement allows you to end a loop or a `switch` statement prematurely. This may be necessary, for example, in order to query errors.

---

```
break [label]
```

---

### Description

The `break` statement has the following functionality, depending on whether it is specified with or without the *label* parameter:

- `break` terminates the execution of the innermost `for`, `for/in`, `while`, `do/while` or `switch` statement, and passes control to the statement which directly follows the loop or the `switch` statement.
- `break label` terminates the nearest statement identified in *label*, and continues processing with the statement immediately after the aborted statement.

If the `break` statement occurs outside a `for`, `for/in`, `while`, `do/while`, or `switch` statement, then a runtime error is reported.

### Example 1

In the following example, the `while` loop is aborted on the fourth pass. The `test` function returns as a result the value of the multiplication  $3 * x$ .

```
function test(x) {  
  var a = 0;  
  while (a < 6) {  
    if (a == 3)  
      break;  
    a ++;  
  }  
  return a * x;  
}
```

*Example 2*

If the user has to enter values in a number of different text boxes in an HTML page, you can use the following loop to check the completeness of the input.

```
bInputComplete = true;
for(field in WT_POSTED)
{
    if WT_POSTED[field] = "" //User has not entered anything
    {
        bInputComplete=false;
        break;
    }
}
if (bInputComplete)
...

```

*Example 3*

If `typeof a[i][j] == 'undefined'`, the program flow continues with the *statement*.

```
start: for ( i=0; i<=99; i++ )
{
    j=0;
    nextloop: for( ; j<=99; j++ )
    {
        // End outer loop if element does not exist
        if ( typeof a[i][j] == 'undefined' )
            break start;
    }
}
statement;

```

### 9.4.8 continue statement

This statement terminates the execution of the statements in a `for`, `for/in`, `while` or `do/while` loop and continues with the next loop cycle.

---

```
continue [label]
```

---

#### Description

The `continue` statement interrupts the execution of the innermost `for`, `for/in`, `while` or `do/while` loop which contains it, and continues processing of the loop as follows:

- If no label is specified, the `continue` statement resumes processing of the innermost loop in which it is contained at the next iteration step.
- If a label is specified, `continue label` resumes processing of the loop identified in *label* at the next iteration step.

In the case of a `while` loop, processing continues with the evaluation of the loop condition, whereas in the case of a `for` loop, it continues with the processing of `update`. If the loop condition is true, the loop is executed again. In the case of a `do/while` loop the end condition is checked again. In the case of a `for/in` loop the next pass is started.

If the `continue` statement occurs outside of a `for`, `for/in`, `while` or `do/while` loop then a runtime error is reported.

#### Example 1

You can skip the processing of fields in which the end user has made no input (`null` value) by querying whether the elements of the posted object have the value `null`:

```
for(i in WT_POSTED){
    if (WT_POSTED[i] == null)
        continue;
    ... // Processing continues here
}
```

#### Example 2

The following loop processes all user input sequentially. All empty entries are ignored.

```
nCount=3;
var eingabe;
for(field in WT_POSTED)
{
    if (WT_POSTED[field] = "") //User has not entered anything
        continue;
    eingabe[i++]=WT_POSTED[field];
    ...
}
```

```
}
```

### *Example 3*

This example finds out the country which the specified city belongs to.

```
cities = new Object;
cities.Germany = new Array ("Aachen", "Bonn", "Essen", "Frankfurt", "Munich,
"Wurzburg");
cities.GreatBritain = new Array
("Birmingham", "Exeter", "Glasgow", "Hull", "London", "Warwick");
cities.France = new Array ("Bourges", "Cannes", "Nantes", "Orleans", "Paris",
"Rennes");
function where(name)
{
  outer: for (country in cities)
  {
    for (i=0; i< cities[country].length; i++)
    {
      if (cities[country][i] == name)
        return country;
      // If the array element is lexically greater
      // than the name we look for, we can continue with
      // the next country.
      else if (cities[country][i] > name)
        continue outer;
    }
  }
  return "";
}
```

## 9.5 return statement

This statement allows a function to return the result of its execution to the calling WebTransactions application component.

---

```
return [ retValue ]
```

---

*retValue* Expression which is passed as the return value

### Description

The `return` statement terminates the execution of a function. If *retValue* is present and is an expression, then it is evaluated and its value returned. If the `return` statement occurs outside of a function, a runtime error is reported.

#### Example 1

The function below calculates the number of hours normally worked by an employee depending on the number of working days in a month and the agreed number of hours per day. The result can be specified as a value or an expression.

```
...
function worktime(days, hours)
{
    workhours = days * hours;
    return (workhours);
}
...
```

or

```
function worktime(days, hours)
{
    return (days * hours);
}
```

#### Example 2

The result of calling the function `square` with parameter `3` is assigned to the variable `result`. The result is output with the `document.write` method.

```
...
function square(x) {
    var y=x * x;
    return y ;
}
result= square(3);
document.write("The square of 3 is " + result);
...
```

## 9.6 var statement

You can create a variable by simply assigning a value (implicitly) or by declaring it with the keyword `var` (explicitly). These two ways of declaring a variable differ only within functions. In a function, an explicit declaration creates a local variable which is only available within this function. In contrast, an implicit variable declaration creates global variables both inside and outside the function and these are valid for the entire template (up to the last `OnReceive` processing step).

---

```
var { identifier | identifier=value } [{ ,identifier | ,identifier=value } ... ]
```

---

`var` Keyword for an explicit variable declaration

*identifier*

Name of the variable. The rules applying to name elements govern the formation of variable names (see [section “Name elements” on page 42](#)).

*value*

Assignment of an initial value, i.e. initialization of the variable. The variable is assigned this value and the corresponding type. If you do not specify a value, the variable is automatically assigned the value `undefined`. This means that nothing is stored.

### Description

You must make an explicit variable declaration whenever you want to declare a local variable within a function. This is because a simple assignment automatically creates a global variable.

You can declare multiple variables together. To do this, separate the variable names with commas.

JavaScript can only handle “loose typing”, i.e. the variable type is not specified on declaration. Data types are automatically converted at runtime if necessary.

#### Example 1

```
...
function set1()
{
    var x = 17;    // Local variable declaration
}
function set2()
{
    document.write("The value of x is " + x);
    // of no use since x is not recognized in this function,
    //Output: The value of x is undefined
}
```

```
}
```

### *Example 2*

```
function set1()
{
    x = 17;    //Global variable declaration
}
function set2()
{
    document.write("The value of x is " + x);

    // Output: The value of x is 17
}
set1();
set2();
...
```

### *Example 3*

```
function calculate(transfer1, transfer2)
{
    var result = transfer1 + transfer2;
    return result;
}
```

**The result of the `calculate` function therefore depends on the types of the transfer parameters. If they are both of type `number`, then the result is also of type `number`. However, if one of the parameters is of type `string`, then a string is returned.**

`calculate(4,7);`      **returns the result 11**

`calculate("4",7);`    **returns the result '47'**



## 9.7 function statement

You use functions to group together sequences of statements which recur within a template. Defining a function allows you to use a simple statement to trigger complex actions without having to write out the same statements each time.

The `function` statement allows you to define your own functions as opposed to the predefined functions which you can call without first having to define them yourself. The functions of built-in classes are described in the [chapter “Built-in classes and methods” on page 117](#).

---

```
function [ identifier ] ( [ parameter [ , parameter ]... ] ) { [ statement... ] }
```

---

### *identifier*

Name used to call (execute) the function. The rules for name elements apply to *identifier* (see [section “Name elements” on page 42](#)).

You can also use functions anonymously by directly assigning the function definition to a variable.

### *Example*

```
erg = function(x)
{
    x += 42;
}
```

### *parameter*

Values which are passed from the calling WebTransactions application component to the function to be processed. *parameter* can be a simple name element (see [section “Name elements” on page 42](#)). The value of the variable is passed, not its address (“call-by-value”). If the function modifies the value of a parameter, this change does not apply to the calling WebTransactions application component.

### *Example 1*

If a function is called with fewer parameters than are specified in its definition, the other parameters are undefined.

```
function myFunc(a,b)
{
    document.write(a + " " + b);
}
myFunc("Hello"); // Output: Hello undefined
```

*Example 2*

If, on calling a function, you specify more parameters than are present in the function's definition, then you cannot access the additional parameters by name. However, you can access them in the function via the `arguments` array.

If you define the function `myfunc` as follows:

```
myfunc(word1,word2)
{
    ...
}
```

and call it as follows

```
myfunc("hello","brave new","world");
```

then, within the function, you can access the 1st parameter via `word1` or `arguments[0]` but you can only access the third parameter via `arguments[2]`.

*statement ...*

A sequence of statements defines the execution logic of the function.

**Description**

The declaration gives the function a name and specifies the statements which are to be executed and the parameters which are to be passed to the function. To call a function, specify the name of the function followed by its parameters enclosed in round brackets. If no parameters are passed, the round brackets are empty.

`function` statements can only be used within functions. The `function` statement generates a local variable of the type `function` in the surrounding function.

*Example*

```
function f()
{
    function g(x)
    {
        return 6*x;
    }
    return g(7);
}
```

The function `g` is no longer valid outside the function `f`. The `function` statement is thus equivalent to the assignment of a function literal to a local variable (see section below):

```
function f()
{
    var g = function (x){return 6*x;};
    return g(7);
}
```

```
}
```

Functions can not only accept but also return values (see [section “return statement” on page 294](#)).

For examples illustrating how functions are declared and called, refer to the [section “return statement” on page 294](#).

## 9.8 Function literal

Alongside the `function` statement (see [page 297](#)), the anonymous use of functions without identifiers is supported - the function literal.

A function literal can be used as part of an expression in a number of places within a script and can even be used in the evaluation operator. As a value, it returns a function which you can assign to a variable and execute .

---

```
function ( [ parameter [ , parameter ]... ] ) { [ statement ... ] }
```

---

*parameter* ...

Values that are passed from the calling WebTransactions application component to the function for processing. *parameter* is a simple name element (see [section “Name elements” on page 42](#)). The value of the variable is passed, not the address of the variable ("call-by-value"). If the function changes the value of the parameter, this change does not apply in the calling WebTransactions application component.

*statement* ...

A sequence of statements defines the execution logic of the function.

*Example*

```
sq = function(x){return x*x;};  
a=sq(5);
```

## 9.9 with statement

This statement defines an object as the default object for statements. This permits a shortened notation for statements with object references.

---

```
with ( object ) block
```

---

*object* Expression which is evaluated and converted into an object

*block* Individual statement or statement block

### Description

Names are interpreted relative to this object in *block*. Nested `with` statements are interpreted from the inside out.

### Example

If you want to assign posted values to a number of different attributes of the system object, then the `with` statement allows for a compact notation:

```
...
with(WT_SYSTEM)
{
COMMUNICATION_ERROR_FORMAT = "wtstart";
STYLE                       = WT_POSTED.STYLE;
LANGUAGE                     = WT_POSTED.LANGUAGE;
TIMEOUT_APPLICATION         = WT_POSTED.TIMEOUT_APPLICATION;
TIMEOUT_USER                 = WT_POSTED.TIMEOUT_USER;
...
}
```

## 9.10 Exception handling

The errors that occur in WebTransactions can be subdivided into the following error classes:

- syntax errors (e.g. incorrect keyword)
- runtime errors (e.g. object not available or array index outside the permitted range)
- communication errors (e.g. connection setup failed)
- global errors (e.g. template not available)

Exception handling provides a very simple means of dealing with errors that occur during execution of a WTScrip script (runtime errors). Through appropriate procedures, it allows you to intercept fatal program errors or other exceptional situations.

### 9.10.1 Error object

Each time a runtime error occurs, an exception is thrown. This exception is an object of type `Error`.

Error objects are structured as follows:

Attribute name	Contents
type	String which identifies the error type The following error types have been defined: <ul style="list-style-type: none"> <li>– <code>CommunicationError</code></li> <li>– <code>GlobalError</code></li> <li>– <code>RTSError</code></li> <li>– <code>JavaError</code></li> </ul>
msgNo#	Number of the error message as it appears in the message file
text	Full text of the error message as it appears in the message file
position	Position at which the error occurred (optional)
position.line	Line in which the error occurred
position.col	Column in which the error occurred
position.path	Name of the template in which the error occurred

It is possible for programmers to catch these exceptions by means of a `try/catch` block, and instigate their own exception handling procedures. If exceptions are not caught, the default functionality comes into play, in which case the error messages are output with the help of the error template.

If an exception is thrown within a section of code, processing of that code section is aborted and no result is returned. All calling code sections are skipped until the program flow reaches a section of script which catches and handles the exception. Once this code section has been processed, i.e. the exception has been handled, the exception is reset and has no affect on the rest of the program flow.

There are two types of exception:

- implicit exceptions
- explicit exceptions

In the event of runtime errors, implicit exceptions are thrown by the WebTA runtime system as instances of the error class. The error class is described in the WebTransactions manual “Concepts and Functions”.

### **Error object in a dynamically generated script**

It is possible for a syntax error to occur not only in a "normal" template, but also in a script created by `eval()` or `setTimeout()`. In this event, additional information is generated to facilitate debugging.

- The child object `Position` of the error object is assigned the attributes `strLine`, `strColumn` and `strText`.  
`strText` contains the string that was dynamically parsed, and `strLine` and `strColumn` refer to this string.
- The values for `strLine`, `strColumn` and `strText` are also output in the WebTransactions trace.
- If the placeholders `%(strLine)`, `%(strColumn)` and/or `%(strText)` are contained in the error template, they are supplied with the current values in the case of errors that can be localized.

## 9.10.2 Explicit exceptions

Explicit exceptions can be thrown in your WTML script using the `throw` statement:

```
throw expression;
```

where *expression* specifies the value of the exception. This can be a literal or an instance of a particular class.

### *Example 1: Literal exceptions*

```
throw 42;                // Generates an exception with the numeric value 42
throw "forty two";      // Generates an exception with the string value "forty two"
throw true;             // Generates an exception with the boolean value "true"
```

### *Example 2: Exception as an object*

```
function UserException (message) // Constructor for object of type UserException
{
    this.message=message;
    this.name="UserException";
}
myUserException=new UserException("Invalid value");
throw myUserException;          // Generates an exception of object type UserException
```



### 9.10.3 Exception handling procedure



Up to Version 6 of WebTransactions, a `try` block without a subsequent `catch` block was possible. As of Version 7, this triggers an error message.

Exception handling allows you to catch predefined implicit or explicit exceptions, and initiate appropriate error handling procedures. For this purpose, WTScript provides the following language elements:

- `try` block
- `catch` block
- `finally` block

The `try ... finally` block is structured as follows:

```
try { guarded code section } ----- (1)
[catch ( identifier if condition ){}] // Multiple catch block ----- (2)
[catch ( identifier if condition ){}]
...
[catch ( identifier ) {}] // Single catch block
[finally{}] ----- (3)
```

#### (1) Guarded code section in the try block

In your script, the code section for which you wish to catch certain exceptions is entered in a `try` block.

#### (2) Error handling in the catch block

The `try` block must be followed by one or more `catch` blocks. This is where exceptions thrown when processing the `try` block are actually handled. Within the `catch` block, *identifier* identifies the exceptions to be handled and exists only while this `catch` block is being processed.

A distinction is made between a single `catch` block and a multiple `catch` block. Since the `try` block can throw various types of exception, you can respond with multiple `catch` blocks which catch each individual exception. It is also possible to attach a `catch` block without any conditions (unconditional `catch` block) to the multiple `catch` block, which will then catch any unexpected exceptions.

**(3) finally block**

The `finally` block contains all statements that are to be executed regardless of whether or not an exception is thrown and/or caught.



Thrown exceptions always terminate the execution context of a function, method, or constructor. A `with` statement will restore the standard object, irrespective of whether or not an exception is thrown.

*Example*

```
function throwException(i, j)
{
    if ( i == j )
        throw true;
    if ( Math.round(100 * ( i + j ) ) == 42 )
        throw 42;
    if ( Math.round(100 * ( i + j ) ) == 21 )
        throw "twenty one";
    else {
        excObj = new Object();
        excObj.type = "any sum";
        excObj.sum = Math.round(100 * ( i + j ) );
        throw excObj;
    }
}

try {
    for ( i=0; i<10; i++ )
    {
        throwException(Math.random(), Math.random());
    }
}

catch ( exc if exc == 42 )
{
    document.write("<BR>The sum of " + i + " + " + j + " = " + exc );
}
catch ( exc if exc == "twenty one" )
{
    document.write("<BR>The sum of " + i + " + " + j + " = " + exc );
}
catch ( exc if exc.type == "any sum" )
{
    document.write("<BR>The sum of " +exc.i+ " + " +exc.j+ " = " +
exc.sum );
}
catch ( exc )
{
    document.write("<BR>"+ i + " and " + j + " are equal. That's " + exc );
}
finally
{
    document.write("<BR>following all exceptions");
}
```



---

## 10 Class templates (\*.clt)

Class templates allow you to automate the evaluation of objects of the same type, for example similar host data objects. Instead of always having to write the same statements for each host data object, you define corresponding class templates. A class template is stored in a file with the suffix `.clt` (class template).

When a class template is evaluated, the result of the evaluation is written to the output stream of the calling template in place of the class template call. In class templates, you can use the same language components as in all other templates. In the class template, the calling object is referenced as `WT_THIS` and its name is accessed with `##WT_THIS#` or `WT_THIS.toString()` (see [page 310](#)). On interpretation, each occurrence of `WT_THIS` is replaced by the calling object.

Like normal templates, class templates are normally stored in the directory `basedir/config/forms`. You can also define different styles and languages for class templates, and the same search strategy is used as for normal templates.

There are two different ways of calling class templates:

- implicitly
- explicitly

### Implicit call

A class template is called implicitly if the evaluation operator is used on a host data object or if the `toString` method is called without an argument for a host data object:

---

```
## hostobject #  
hostobject.toString()
```

---

In the case of an implicit call, the type of the host data object is determined and used to form the name of the class template: `type.clt`.

The `type` is determined from the communication module. In the case of openUTM, the `IOTYPE` attribute is evaluated, and in the case of OSD/MVS, the `Type` attribute.

## Explicit call

When an explicit call is issued, the name of the class template is specified in the `toString` method (see [section “toString method” on page 155](#)). This means that you can define class templates independently of the type of host data object in question and call these as required

---

```
##hostobject.toString(name)#
```

---

The suffix `.clt` is automatically appended by WebTransactions.

## 10.1 WT\_THIS - accessing the calling object

Within a class template, you can use the `WT_THIS` keyword to access the calling object:

`WT_THIS` supplies a reference to the calling object. This allows you, for example, to query and modify the attributes of this object (see [section “Example: class templates and WT\\_THIS” on page 311](#)).

However, the evaluation operator and the `toString / valueOf` methods function differently in `WT_THIS` and in the calling object. They supply the name of the calling object. This distinction is useful since otherwise, expressions such as `##WT_THIS#` would result in the class template calling itself for ever. Furthermore, in the class template it is advantageous to access not only the object but also its name. For example, in this way you can derive names for HTML tags at the interface from the name of the calling object:

```
<INPUT TYPE="TEXT" NAME="##WT_THIS#" VALUE="##WT_THIS.Value#">
```

Within a class template, the value posted by the browser can be queried as follows:

```
WT_POSTED[WT_THIS]
```

## 10.2 Example: class templates and WT\_THIS

The example below illustrates a class template named `INPUT.clc` used for text boxes in `openUTM`.

The section treats protected fields that are displayed on the page as text. The display attributes of each output field are checked and the display is customised on the generated HTML page. For example, if the field requires a color display, a font tag with the corresponding color is generated around the text. If it needs to be highlighted, (`wt_this.Intensity == 'H'`), it will be included on the generated HTML page also in `<b>`.

```

if ( wt_this.Visible == 'Y' )
{
    output = wt_this.HTMLValue;
    if ( wt_this.Inverse == 'Y' )
    {
        if (wt_this.Color && wt_this.Color.toUpperCase() != 'N' && wt_this.Color
        != ' ' )          output = '<font COLOR=#000000 STYLE=\"background-color:'
        +colors[wt_this.Color-1] + '\">' +output+'</font>';
    }
    else if (wt_this.Color&&wt_this.Color.toUpperCase() != 'N' && wt_this.Color
    != ' ' )          output = '<font color=' + colors[wt_this.Color-1] + '>' +
    output + '</font>';
    if (wt_this.Intensity == 'H')
        output = '<b>' + output + '</b>';
    if (wt_this.Blinking == 'Y')
        output = '<i>' + output + '</i>';
    if (wt_this.Underlined == 'Y')
        output = '<u>' + output + '</u>';
    document.write( output );
}

```





---

# 11 Master templates (.wmt)

Master templates are used by WebTransactions when generating the Automask and the format-specific templates. Their purpose is to guarantee a uniform layout.

Like all other templates, they can contain fixed HTML areas as well as any number of WTML tags or WTScrip scripts. They may also include special master template tags, known as MT tags for short. The individual MT tags are described in sections [“Lines tag” on page 315](#) through [“GlobalSettings tag” on page 331](#).

All master template file names have the suffix `.wmt`.

## Strengths of the master template concept

The strengths of the master template concept are particularly evident in host applications containing numerous formats with a similar layout, e.g. comprising a header, a work area, and a footer. All you need to do is define the layout once in the master template, and apply the master template when generating the format-specific templates. All generated templates are then automatically assigned the desired layout.

This not only ensures consistency (corporate look and feel), but also reduces the development costs for your WebTransactions applications. Any changes or adaptations to the formats of your host applications can be made quickly and easily at a central location.

## Standard master templates

Each of the supply units WebTransactions for OSD, WebTransactions for MVS and WebTransactions for openUTM is supplied with its own standard master template, which can be applied as is or customized if desired. The standard master templates contain all WTML tags and WTScrip scripts which are common to all templates of the respective supply unit, e.g. a check to establish whether or not a private system object exists.

The standard templates for the different supply units are described in detail in the relevant manuals.

### Using the master template

In the WebLab graphical user interface, you specify the master template to be used for generation. By default, this is preset to the standard master template of the relevant supply unit.

Some generation options (e.g. the generation method or display attributes) can also be defined directly in WebLab. If the generation option is already set in the master template, then this setting is displayed by default. The value set with WebLab is always used for these options and applies to the entire template. If you change a value with WebLab, then this overrides the value set in the master template.

If you set one or more options more than once in the master template, the last value set applies.

### Syntax of MT tags

MT tags begin and end with a percentage sign. The string %% is reserved for MT tags.

Some MT tag names may be followed by a parameter list. Neither the tag names nor the parameter names are case-sensitive. If a parameter is set a number of times, the last value set applies.

The individual syntax elements may be separated by blanks.

All MT tags are interpreted even if they are located within HTML or WTML comments.

## 11.1 Lines tag

The `Lines` tag defines where and how format-specific sections are to be generated.


---

```
%%Lines parameters%
```

---

The individual parameters are described in the table below:

Parameter	Meaning	Possible values	Default
<code>LayoutOnly</code>	Enables the implementation of functions/methods for transferring the input parameters.	<b>Yes</b> Only the layout of the host objects in the template is generated. Any further action such as generation of the <code>wtInputFields</code> object is suppressed. <b>No</b> Full function scope as previously: Variables and statements for transferring inputs are generated.	No
<code>Breaks</code>	Controls generation of screen layout with static display attributes and static text.	<b>Yes</b> Line feed marks are generated after each line field. The corresponding field content is also displayed on a comment line. <b>No</b> All fields of a screen line are displayed in the template on one line. Comments with the field contents at the moment of capture are not generated.	Yes
<code>PartialFormat</code>	Specifies the master template for partial or full format generation.	<b>Yes</b> Master template for partial format generation <b>No</b> Master template for full format generation	No
<code>StartLine</code>	First format line to be generated.	Integer	1

Parameter	Meaning	Possible values	Default
EndLine	Last format line to be generated. WebTransactions checks whether EndLine is smaller than StartLine. If it is, the MT Lines tag is ignored.	Integer	Last Line
StartPattern	The first format line to be generated is that immediately after the line containing the specified string.	String	None
EndPattern	The last format line to be generated is that immediately before the line containing the specified string.	String	None
CellsDelimiter	Cell delimiter.	String (individual character). If all the characters of a format line correspond to the character specified in CellsDelimiter, the string is replaced by that contained in the CellsDelimiterReplace parameter.   This parameter is only effective for fields of type FIXTEXT.	
CellsDelimiterReplace	String used for replacement	String used for a line in which all the characters correspond to the value of CellsDelimiter.	HTML tag for a new table cell
CursorInProtectedField	Position the cursor in protected fields.	<b>Yes</b> The cursor can be positioned in protected fields of the generated templates. <b>No</b> The cursor cannot be positioned in protected fields of the generated templates.	No

Parameter	Meaning	Possible values	Default
Generate	Generation method.	<p><b>Class</b> Input/output processing of host data objects is based on class templates, i.e. each host data object is addressed via a class template (INPUT.clt or OUTPUT.clt). This method is recommended if global settings are to be defined for input/output fields. If corresponding class templates do not yet exist, they are created automatically.</p> <p><b>Inline</b> All the steps required for HTML representation and the processing logic are contained in the generated template in HTML tags, WTML tags, and WTScrip scripts. In other words, none of the processing steps are relocated to user exits or class templates. This is the most flexible option if you wish to edit your templates later on.</p>	Inline
DisplayAttributes	Support for field display attributes.	<p><b>Dynamic</b> If this option is selected then the functions taggedInput() for entry field generation and taggedOutput() for output field generation may be used. See also the taggedInput and taggedOutput parameters.</p> <p><b>Static</b> The display attribute values are fixed in the generated template code as defined in the FLD file or recorded during the capturing procedure.</p> <p><b>No</b> Display attributes are not supported.</p>	No

Parameter	Meaning	Possible values	Default
StaticText (OSD and MVS only)	Conversion of protected fields into static HTML text.	<p><b>Yes</b> If this option is enabled, the text contained in protected format fields (output fields of type "Protected") is generated directly as text in the HTML data stream. This assumes that protected fields are generally static, i.e. not supplied with variable values by the host application. If certain protected fields are variable, then you must set the corresponding values manually, i.e. you must program the templates accordingly. You should use this option if only a few protected fields of a format contain variable text.</p> <p><b>No</b> Protected fields are evaluated dynamically by WebTransactions.</p>	No
DisplayEuro	Display of the euro symbol.	<p><b>Yes</b> If the format contains fixed text with the euro symbol (previously a currency symbol), this is converted to the HTML escape sequence <code>&amp;#8364</code> (euro).</p> <p><b>No</b> The previous currency symbol in fixed text is converted to X'A4' (¤) as opposed to the euro symbol.</p>	No
MenuBar (MVS only)	Conversion of the first format line into a menu bar.	<p><b>Yes</b> Selectable fields in the first line of the format are interpreted as menu commands and converted to pull-down menus. This option is recommended for ISPF applications.</p> <p><b>No</b> The first format line is converted as normal.</p>	No

Parameter	Meaning	Possible values	Default
TaggedInput	Use of the <code>taggedInput()</code> function	<p><b>Disabled</b> The <code>taggedInput()</code> function is not used in the generated templates and is not defined in the master template.</p> <p><b>Enabled</b> The <code>taggedInput()</code> function is used in accordance with the display attributes which you have specified in the master template or in WebLab. You should note that the <code>MT Lines tag</code> may result in the generation of additional HTML tags.</p> <p><b>Enforced</b> Only the <code>taggedInput</code> function is used on template generation. The <code>MT Lines tag</code> does not generate any additional HTML tags.</p>	Disabled
TaggedOutput	Use of the <code>taggedOutput()</code> function	<p><b>Disabled</b> The <code>taggedOutput()</code> function is not used in the generated templates and is not defined in the master template.</p> <p><b>Enabled</b> The <code>taggedOutput()</code> function is used in accordance with the display attributes which you have specified in the master template or in WebLab. You should note that the <code>MT Lines tag</code> may result in the generation of additional HTML tags.</p> <p><b>Enforced</b> Only the <code>taggedOutput</code> function is used on template generation. The <code>MT Lines tag</code> does not generate any additional HTML tags.</p>	Enabled

### Interaction between start and end conditions

Beginning with the first line that fulfills a start condition (`StartLine` or `StartPattern`), the format is converted line by line. The conversion process ends with the first line that fulfills an end condition (`EndLine` or `EndPattern`), irrespective of the parameter specified for the start condition.

#### *Example*

```
%%Lines
StartLine = 4
StartPattern = "Options"
EndPattern = "End"
EndLine=15
%
```

Let's assume that the first occurrence of the string "Options" occurs in line 2 of the format and the first occurrence of the string "End" occurs in line 18. In this case, the converted area extends from line 3 to line 15.

`WebTransactions` checks whether `EndLine` is smaller than `StartLine`. If it is, the `MT Lines` tag is ignored.



## 11.2 Options tag

With the `Options` tag you can define both values for standard master template tags and also individual master template tags. A standard and extended syntax are available for this purpose.

### 11.2.1 Options tag (standard syntax)

With standard syntax you can use the parameters of the `Options` tag to specify global values, which are then used in all places in which there is a tag corresponding to the parameter. No code is generated in the templates for the `Options` tag itself.

If a master template contains several `Options` tags that set the same parameter, the last value set explicitly for the corresponding tag applies.

---

```
%%Options parameters%
```

---

The individual parameters are described in the table below:

Parameter	Meaning	Possible values	Default
JavaUtil	Contains the name of the Java path for the Java utility package *	String	java.util
CommObj	Reference to the communication object.	String	Depends on the protocol used: OSD_0, MVS_0 or UTM_0
NationalVariant	Language variant used for messages exchanged between WebTransactions and the host application.	"International" "English (UK)" "English (USA)" "Swedish" "German" "French" "Italian" "Spanish" "Swiss" "Norwegian" "Danish" "French-Belgian"	"International"

To ensure that a value set in the `Options` tag is taken into consideration, the parameter must be set **before** you use the corresponding tag.

*\* Notes*

1. The entries for `JavaUtil` are only meaningful in the master template `JAVA_B0.wmt`.
2. The settings are used implicitly when the `ObjectCreate` tag is replaced to reference the different Java classes from which the objects are recruited.

*Example*

- Master template code:

```
%%NationalVariant%  
%%Options NationalVariant = "Italian"%  
%%NationalVariant%  
%%Options NationalVariant = "Spanish"%  
%%NationalVariant%
```

- Generated code:

```
International //(Default)  
Italian  
Spanish
```

## 11.2.2 Options tag (extended syntax)

You can use the extended syntax of the options tag to define your own master template tags. This may, for example, be useful if you want to use a piece of text several times in the master template or if you want to generate different texts, according to whether the master template is used to generate an automask or a format-specific (individual) template.

---

```
%%Options
    destination="Automask | Individual | Both"
    tagName=" tagname"
block
%
```

---

*destination*

The template type is indicated for the generation of which the self-defined tag is valid.

*tagname*

Name of self-defined tag. Case-sensitive.

*block*

Text area that is marked as replacement text for the tag.

*Instruction*

The % end character must be specified as a single character on its own line. This ensures that other master template tags can also be used inside a block.

When formulating the *block* text note the following:

1. The text must fit into the environment in which you use the self-defined tag. This means, for example: If a self-defined tag is to be used in a WTScrip area, you can only use WTScrip in this tag.
2. In the *block* text you may use only the tags %%CommObj%, %%NationalVariant%, %%Format%, %%Source%, %%ObjectName%, %%PackageName%, %%JavaUtil%, and the tags that you have yourself defined. These nested tags are replaced by the content when the outer tag is replaced.

You use the self-defined master template tags in the form

```
%%tagname%
```

The *tagname* that you specify must be exactly the same as the name that you have defined for the `tagName` parameter (case-sensitive!).

*Examples***Example 1:**

Display of an image that varies according to whether the template used is an automask or a format-specific template.

**master template:**

```

...
%%GenerationInfo%
%%Rem Use fig.1 for individual templates%
%%Options tagName="Image" destination="Individual"

%
%%Rem Use fig.2 for Automask%
%%Options tagName="Image" destination="Automask"

%
...
<body bgcolor="#C0C0C0">
%%Image%
<form WebTransactions name="%%Format%">

```

**Code generated in the automask:**

```

<body bgcolor="#C0C0C0">

<form WebTransactions name="Automask">

```

**In format-specific template generated Code:**

```

<body bgcolor="#C0C0C0">
<form
WebTransactions name="Trav0">

```

**Example 2:**

Different generation of the `taggedInput` function in order to avoid redundant queries in the automask and thereby improve its performance.

**Extract from the master template:**

```

%%Rem for individual templates. You must also once again check that
a field that was an input field at the moment
of capture is still an input field.

%
%%Options destination="Individual" tagName="taggedInputCrossCall"
    if ( hostObject.Type == 'Protected' && hostObject.Markable == 'No' )
        return taggedOutput( hostObject );
%
%%Rem This query is not required in the automask%
%%Options destination="Automask" tagName="taggedInputCrossCall"
%

function taggedInput( hostObject )
{ %%taggedInputCrossCall%

currentLength = hostObject.Length;
input = '<input type=' + (hostObject.Visible == 'No' ? '"password"' : '"text"' );
if ( WT_BROWSER.is_ie || WT_BROWSER.is_ns61up)
{
input += ' class="box" style="width:' + (currentLength * WT_BROWSER.charWidth + 1) +
'px';
input += (hostObject.Blinking == 'Yes' ? ' ; background color:#FFC0C0' : '');
input += (hostObject.Underline == 'Yes' ? ( hostObject.Intensity == 'Reduced' ? ' ;
color:#A0A0FF' : ' ; color:#0000A0' ) :( hostObject.Intensity == 'Reduced' ?
' ; color:#A0A0A0' : ' ' )) + '";';
}

input += ' name="' + hostObject.Name + '" size="' + currentLength
+ '" maxlength="' + currentLength
+ '" value="' + hostObject.Value
+ (hostObject.Input == 'Numeric'?'" numeric="1':"')
+ (hostObject.Markable == 'Yes'?'" markable="1':"')
+ '">';

document.write( input );
}

```

**Extract from the generated automask (query not generated):**

```
function taggedInput( hostObject )
{
currentLength = hostObject.Length;
input = '<input type=' + (hostObject.Visible == 'No' ? "password" : "text" );
if ( WT_BROWSER.is_ie || WT_BROWSER.is_ns61up)
{
input += ' class="box" style="width:' + (currentLength * WT_BROWSER.charWidth + 1) +
'px';
input += (hostObject.Blinking == 'Yes' ? ' ; background-color:#FFCOCO' : '');
input += (hostObject.Underline == 'Yes' ? ( hostObject.Intensity == 'Reduced' ? ' ;
color:#A0A0FF' : ' ; color:#0000A0' ) :( hostObject.Intensity ==
'Reduced' ?
' ; color:#A0A0A0' : ' ' )) + '";';
}
input += ' name="' + hostObject.Name + '" size="' + currentLength
+ ' maxlength="' + currentLength
+ ' value="' + hostObject.Value
+ (hostObject.Input == 'Numeric'? " numeric="1':"')
+ (hostObject.Markable == 'Yes'? " markable="1':"')
+ '"/>';
document.write( input );
}
```

**Extract from an individual template (query generated):**

```
function taggedInput( hostObject )
{
if ( hostObject.Type == 'Protected' && hostObject.Markable == 'No' )
return taggedOutput( hostObject );
currentLength = hostObject.Length;
input = '<input type=' + (hostObject.Visible == 'No' ? "password" : "text" );
if ( WT_BROWSER.is_ie || WT_BROWSER.is_ns61up)
{
input += ' class="box" style="width:' + (currentLength * WT_BROWSER.charWidth + 1) +
'px';
input += (hostObject.Blinking == 'Yes' ? ' ; background-color:#FFCOCO' : '');
input += (hostObject.Underline == 'Yes' ? ( hostObject.Intensity == 'Reduced' ? ' ;
color:#A0A0FF' : ' ; color:#0000A0' ) :( hostObject.Intensity ==
'Reduced' ?
' ; color:#A0A0A0' : ' ' )) + '";';
}
input += ' name="' + hostObject.Name + '" size="' + currentLength
+ ' maxlength="' + currentLength
+ ' value="' + hostObject.Value
+ (hostObject.Input == 'Numeric'? " numeric="1':"')
+ (hostObject.Markable == 'Yes'? " markable="1':"')
+ '"/>';
document.write( input );
}
```

## 11.3 Rem tag

This tag appears only in the master template and enables the master template developer to comment the master template without the comment lines being transferred to the generated template.

---

```
%%Rem comment%
```

---

The individual parameters are described in the following list:

<b>Parameter</b>	<b>Meaning</b>	<b>possible values</b>	<b>default value</b>
<i>comment</i>	Insert comment	Any string	none

## 11.4 OnReceiveCopies tag

OnReceiveCopies tags can only be used in WTScrip areas (and make sense only within OnReceiveScript areas). In generated templates, they are replaced by statements which transfer the values received from the browser to the corresponding host data objects.

---

```
%%OnReceiveCopies parameters%
```

---

The individual parameters are described in the table below:

Parameter	Meaning	Possible values	Default
StartLine	First line of the format area for whose fields posted values are to be transferred to host data areas.	Integer	1
EndLine	Last line of the format area for whose fields posted values are to be transferred to host data areas.	Integer	Last Line
StartPattern	The first line of the format area for whose fields posted values are to be transferred to host data areas is that immediately after the line containing the specified string.	String	None
EndPattern	The last line of the format area for whose fields posted values are to be transferred to host data areas is that immediately before the line containing the specified string.	String	None

### Example

- Master template code:

```
<wtOnReceiveScript>
<!--
host.WT_FOCUS.Field = WT_CURSOR;
%%OnReceiveCopies%
/-->
</wtOnReceiveScript>
```

- The generated code could look like this:

```
<wtOnReceiveScript>
<!--
host.WT_FOCUS.Field = WT_CURSOR;
// ***** copy posted values to host objects *****
host1.E_05_025_001.Value = WT_POSTED.E_05_025_001;
/-->
</wtOnReceiveScript>
```



## 11.5 GenerationInfo tag

The `GenerationInfo` tag creates an WTML comment containing information on the generation options and the circumstances surrounding the generation process.

It can be specified in the fixed HTML area, within WTML tags, or within WTScrip areas.

---

```
%%GenerationInfo%
```

---

This tag creates the following lines, for example:

```
<wtrem>*****</wtrem>
<wtrem>** WTML document: AutomaskOSD **</wtrem>
<wtrem>*****</wtrem>
<wtrem>**
<wtrem>** Document generation based on Master Template : **</wtrem>
<wtrem>** C:\Program Files\webtransactions\75\weblab\OSD.wmt **</wtrem>
<wtrem>**
<wtrem>** Generated at Tue Jun 08 12:44:20 2010 **</wtrem>
<wtrem>**
<wtrem>** Options used by the generator : **</wtrem>
<wtrem>** - %OPTIONS: **</wtrem>
<wtrem>** CommObj = OSD_0 **</wtrem>
<wtrem>** NationalVariant = International - PartialFormatMode = No **</wtrem>
<wtrem>** - %OPTIONS: **</wtrem>
<wtrem>** self defined Tag = taggedInputCrossCall **</wtrem>
<wtrem>** - %OPTIONS: **</wtrem>
<wtrem>** self defined Tag = taggedInputCrossCall **</wtrem>
<wtrem>** - %OPTIONS: **</wtrem>
<wtrem>** self defined Tag = taggedOutputCrossCall **</wtrem>
<wtrem>** - %OPTIONS: **</wtrem>
<wtrem>** self defined Tag = taggedOutputCrossCall **</wtrem>
<wtrem>** - %LINES: **</wtrem>
<wtrem>** TaggedInput = Enabled - TaggedOutput = Enabled **</wtrem>
<wtrem>** DisplayAttributes = Dynamic - CursorInProtectedField = Yes **</wtrem>
<wtrem>** - %RECEIVES: **</wtrem>
<wtrem>** Parameters not specified **</wtrem>
<wtrem>*****</wtrem>
<wtrem>** WebTransactions V7.5 Fujitsu Technology Solutions 2010 **</wtrem>
<wtrem>*****</wtrem>
```

## 11.6 Format tag

In generated templates, the `Format tag` is replaced by the name of the current converted format.

---

```
%%Format%
```

---

*Example*

```
<wtData form=%%Format%>
```

## 11.7 CommObj tag

In generated templates, the `CommObj tag` is replaced by the string specified in the `CommObj` parameter of the `MT Options tag` (reference to the communication object). The default value depends on the protocol: `OSD_0`, `MVS_0` or `UTM_0`.

---

```
%%CommObj%
```

---

*Example*

Master template code:

```
%%Options CommObj="host1"%  
if (%%CommObj%.WT_SYSTEM!= null)  
host_system = %%CommObj%.WT_SYSTEM;
```

## 11.8 NationalVariant tag

In generated templates, the `NationalVariant tag` is replaced by the string specified in the `NationalVariant` parameter of the `MT Options tag` (language variant used for messages exchanged between WebTransactions and the host application).

---

```
%%NationalVariant%
```

---

## 11.9 GlobalSettings tag

You can use the `GlobalSettings` tag to define the general settings for the master template. The `GlobalSettings` tag can be located anywhere in the master template, but may occur only once. If you use the `GlobalSettings` tag more than once in the master template, then WebLab issues an error message and generation is not started

---

```
%%GlobalSettings parameters%
```

---

The individual parameters are described in the list below:

Parameter	Meaning	Possible values	Default
Protocol	Host application protocol	"OSD" "MVS" "UTM" "JAVA" "WSDL"	None

*Example*

```
%%GlobalSettings protocol="OSD"%
```

## 11.10 Source tag

The `Source` tag is valid only for the master template `Java_BO.wmt`, and contains the name of the copied description file in the base directory (e.g. `WSDL/B01.wsdl`).

---

```
%%Source%
```

---

## 11.11 ObjectName tag

The `ObjectName` tag is valid only for the master templates `WSDL.wmt` and `Java_BO.wmt` and contains the name of the Java object (either the BO or the EJB name) or the name of the service (in the case of web service).

---

```
%%ObjectName%
```

---

## 11.12 PackageName tag

The `PackageName` tag is valid only for the master template `Java_BO.wmt` and contains the name of the Java package in which the object is contained.

---

```
%%PackageName%
```

---

## 11.13 BinaryFile tag

The `BinaryFile` tag is valid only for the master template `Java_BO.wmt`, and supplies the name of the *n*th copied file.

---

```
%%BinaryFile parameters%
```

---

The individual parameters are described in the list below:

Parameter	Meaning	Possible values	Default
Number	Name of the <i>n</i> th copied binary file	File name, e.g. JAVA/BO1	1

*Example*

```
%%BinaryFile Number=2%
```

## 11.14 ArchiveName tag

The `ArchiveName` tag is valid only for the master template `Java_BO.wmt` and contains the name of the Java archive.

---

```
%%ArchiveName%
```

---

## 11.15 MethodInterface tag

The `MethodInterface` tag is valid only for the master templates `WSDL.wmt` and `Java_BO.wmt`, and is used to generate the display of individual methods.

---

```
%%MethodInterface%
```

---



---

## 12 Server-side interfaces - Java integration and user exits

It is possible to extend the functionality of your WebTransactions application as follows:

- Thanks to the integration of Java in WebTransactions, you can instantiate your own Java class in WebTransactions, use the methods of the resulting objects, and access the object attributes.
- With user exits, you can integrate your own C/C++ functions in WebTransactions.

You could, for example, instigate access to structured data (tables, lists) in a user exit, or determine the coordinates of a graphic to be embedded in an HTML page.

You can also use user exits, for example, for the dynamic construction of HTML pages without connecting a host application. You might, for example, read the data for presentation from a file. In this type of application, WebTransactions can be thought of as a convenient way into the CGI interface.

Examples of this type of application are:

- visitors' book functions
- management and polling of telephone lists
- administrative interface for the HTTP server. In this case, user exits must read and write the http daemon configuration file and restart the daemon.

WebTransactions provides a set of ready-made C/C++ user exits. Some of these are used internally by WebTransactions - e.g. in the start templates - but they can also be integrated in customized templates or in templates you have created yourself. The interfaces of these user exits are described in [section "Ready-made C/C++ user exits supplied with WebTransactions" on page 355ff.](#)

## 12.1 Java integration in WebTransactions

Thanks to the integration of Java in WebTransactions, you can instantiate any Java classes in your WebTransactions application in order to use the methods of the resulting objects and access the object attributes.

To ensure that Java programs or applications will run on your WebTransactions system, a Java runtime environment (Java Virtual Machine, JVM) must be installed on this system.

Please note the following regarding Java integration:

- Decimal numbers exchanged between WTSript and Java will change their representation. For example, the value 2.1 becomes 2.099999904632568 after it is transferred to Java and back again.
- WTSript does not support any Unicode characters. The standard conversion of strings at the Java interface always converts from ISO 8859-1 to UTF 16 and vice versa. As of WebTransactions V6 this conversion can be controlled by the variable `WT_SYSTEM.JAVA_CHARSET`. It must be set before the first `WT_JAVA` access. Its value indicates a runtime library (name without `.so` or `.dll`) that contains the appropriate conversion functions. This runtime library is searched for in the base directory and in the installation directory under `lib`.

The standard version contains a runtime library for the character set ISO 8859-2 (Central Europe). Other libraries can be supplied on request.

### *Example*

```
<html>
<head>
<title>character set Test</title>
</head>
<body>
<H1>Character Set Test</H2>
<pre style="font size: 16pt">
ISO 8859-2

<wtoncreatescript>
<!--
// convert Browser and Java to ISO 8859-2
WT_SYSTEM.CHARSET = WT_SYSTEM.JAVA_CHARSET = 'ISO 8859-2';

//character set output
hex = '0123456789ABCDEF';
document.writeln (' <b>0 1 2 3 4 5 6 7 8 9 A B C D E F</b>');
for (i=0; i<16; i++)
{
```



```

document.write ('<b>',hex[i], '</b>');
if (i<2 || i==8 || i==9)
{
    document.writeln();
    continue;
}
for (j=0;j<16; j++)
    document.write (' ', eval ('"\x'+hex[i]+hex[j]+'"));
document.writeln();
}

```

```

// generate Java string
a = new WT_JAVA.java.lang.String (WT_POSTED.s||'');
//-->
</wtoncreatescript>

```

<form webtransactions>

In order to test correct processing in Java, characters can be input in the following input field and be posted to WebTA (e.g. by Copy/Paste from the character matrix above) They are then converted into a Java string, converted toUpperCase and toLowerCase using the Java methods and then converted back to WebTA strings and displayed.

```

<input type="text" name="s"> <input type="submit" value="convert">
</form>
Enter: ##WT_POSTED.s#
Java toUpperCase: ##a.toUpperCase()#
Java toLowerCase: ##a.toLowerCase()#
</pre>
</body>
</html>

```

## 12.1.1 Installing the Java runtime environment



If you wish to take advantage of Java integration or Java user exits in WebTransactions, the Java runtime environment must be installed **before** WebTransactions.

If you have already installed a Java runtime environment on your WebTransactions system, no further steps are required.

If the Java runtime environment has not yet been installed the following table tells you where you can obtain the software:

<b>Windows</b> <b>Solaris</b> <b>Linux</b>	For 32-Bit Windows platforms, Solaris and Linux, the Java runtime environment (freeware) is supplied by Sun Microsystems. You can download it directly from the Sun Java Software Website ( <a href="http://java.sun.com">http://java.sun.com</a> ).
<b>BS2000/OSD Posix</b>	For BS2000/OSD V3.0 or later, the Java environment JENV (BS2000/OSD) V1.3 is supplied as standard software but may need to be installed and activated for Posix.

## 12.1.2 Activating Java support

WebTransactions initializes a Java Virtual Machine (VM). The Java libraries containing the native Java functions must therefore be accessible. When installing WebTransactions, you will be asked to specify the appropriate path. (For Unix platforms or Windows platforms, the JRE installation directory is the directory in which the `jvm.dll` library is located). WebTransactions then sets the necessary file references automatically.

If WebTransactions cannot load the Java libraries (e.g. because the files have been moved to another directory since WebTransactions was installed or because you are installing Java at a later date), an error message is output. In such cases you must create the file references yourself.

The following sections take JDK 1.3.1 as an example and specify the file references required (depending on the *Webtransaction* platform).

## Windows

In the WebTransactions installation directory, in the subdirectory `lib`, create a file reference `javasys2` to the directory containing the `jvm.dll` library. For example, the reference

```
C:\installdir\lib\javasys2 to the directory C:\jdk1.3.1\jre\bin\classic.
```

## Solaris/Linux

In the subdirectory `lib` of the WebTransactions installation directory, generate the following symbolic links :

```
javalib -> ../jdk1.3.1/jre/lib/i386 (directory containing additional shared objects)
javasys -> ../jdk1.3.1/jre/lib
javathreads -> ../jdk1.3.1/jre/lib/i386/native_threads
javavm -> ../jdk1.3.1/jre/lib/i386/server (directory containing libjvm.so)
```

where “...” stands for the JDK installation directory.

## BS2000 OSD V2.0 or later

During public installation in the Posix environment, the file references are set automatically. In the case of a private installation, you may need to set the file references for `javasys` and `javathreads` explicitly. To do this, create the following symbolic links in the WebTransactions installation directory `/opt/WebTrans/7.5/lib` :

```
javalib -> ../jdk1.3.1/jre/lib/i386 (directory containing additional shared objects)
javasys -> ../jdk1.3.1/jre/lib
javathreads -> ../jdk1.3.1/jre/lib/i386/native_threads
javavm -> ../jdk1.3.1/jre/lib/i386/server (directory containing libjvm.so)
```

“...” stands for the JDK installation directory.

### 12.1.3 Defining parameters for the Java Virtual Machine (JVM)

When defining the parameters of the Java Virtual Machine (JVM) used in WebTransactions, you have the following two options:

- Define the JVM parameters with the help of system attributes
- Define the JVM parameters with the help of the `WT_SYSTEM.JAVA_OPTIONS` array

#### Defining parameters for the Java Virtual Machine (JVM) using system attributes

The following system attributes can be used to control the Java Virtual Machine (JVM):

System attribute	Compiler option for Java	Meaning
<code>JAVA_CHECK_SOURCE='1'</code>		Checks whether the source files have a more recent date than the class files.
<code>JAVA_CLASSPATH</code>	<code>-classpath</code>	Sets the search path for the class files. Multiple directories are separated by semi-colons. Default: <code>basedir/java</code>
<code>JAVA_DEBUG='1'</code>	<code>-debug</code>	Activates remote debugging (only for the debug version of the Java system library).
<code>JAVA_DEBUG_PORT</code>		Port for remote debugging.
<code>JAVA_DISABLE_ASYNC_GC='1'</code>	<code>-noasyncgc</code>	Deactivates asynchronous garbage collection.
<code>JAVA_DISABLE_CLASS_GC='1'</code>	<code>-noclassgc</code>	Deactivates garbage collection.
<code>JAVA_ENABLE_VERBOSE_GC='1'</code>	<code>-verbosegc</code>	Prints a message on each garbage collection.
<code>JAVA_INITIAL_HEAP_SIZE</code>	<code>-ms number</code>	Initializes the Java memory.
<code>JAVA_MAX_HEAP_SIZE</code>	<code>-mx number</code>	Sets the maximum Java memory.
<code>JAVA_NATIVE_STACK_SIZE</code>	<code>-ss number</code>	Sets the maximum memory size for thread processing.
<code>JAVA_STACK_SIZE</code>	<code>-oss number</code>	Sets the minimum memory size for thread processing.
<code>JAVA_VERBOSE='1'</code>	<code>-verbose</code>	Deactivates verbose mode.
<code>JAVA_VERIFY_MODE='0'</code>	<code>-noverify</code>	Does not check classes.
<code>JAVA_VERIFY_MODE='1'</code>	<code>-verifyremote</code>	Only checks classes read via the network.
<code>JAVA_VERIFY_MODE='2'</code>	<code>-verify</code>	Checks all classes.

## Defining parameters for the Java Virtual Machine (JVM) using the WT\_SYSTEM.JAVA\_OPTIONS array

You can use the WT\_SYSTEM.JAVA\_OPTIONS array to transfer all default arguments to the JVM. All parameters for the JVM must be defined before WT\_JAVA is accessed for the first time. It is not possible to make any further changes during the session.

### Example

```
WT_SYSTEM.JAVA_OPTIONS = new Array('-verbose', '-Xms6m');
a = new WT_JAVA.java.lang.String("Hello World!");
```

## 12.1.4 Creating Java objects in WTSript

Java objects are created in WTSript using the `new` operator:

---

```
var foo = new WT_JAVA.classname();
```

---

### *classname*

Name of the Java class to be instantiated. This must be a fully-qualified class name. In the case of constructors without parameters, the parentheses may be omitted.

### Example

```
var myString = new WT_JAVA.java.lang.String("Hello world");
```

Java objects created in this way are handled by WTSript taking into consideration the following special features:

- Only the object itself is displayed in the WebLab object tree, i.e. attributes and methods will not be visible. The object value is set to the result of the `toString()` method executed implicitly by the object.
- It is still possible to access all methods and attributes of the object. This applies both for attributes defined in the associated class and for inherited attributes.



Variables of primitive Java types (`char`, `int`, `double`, etc.) cannot be created as described above. These must be created in WTSript using the associated wrapper class (in this case, `Double`) as follows:

```
var foo = (new WT_JAVA.java.lang.Double(value)).doubleValue();
```

where *value* specifies the value to which the variable is to be set.

The same procedure applies for `char/Char`, `int/Int`, `long/Long`, etc.

## 12.1.5 Using Java objects in WTSript

Java objects can be used in WTSript in exactly the same way as they are used in Java, apart from the following restrictions:

- WTSript objects cannot inherit from Java objects.
- Additional attributes or methods cannot be defined for Java objects in WTSript.
- WTSript has no `cast` operator. This is generally not required, since a check is carried out automatically to establish the types to which a particular value can be converted without loss of data. Casts to other object types, which may be necessary when invoking certain methods (with the help of objects created with `new`) **????**, are carried out automatically provided they are permitted by Java rules. In the process, even primitive Java data types are converted. In contrast to normal Java behavior, however, some types may also be converted to “lesser” types (e.g. `int` → `short`) if this is possible without loss of data.

To access class elements (i.e. attributes and methods) in WTSript, you use:

---

```
var foo = objectname. $\left. \begin{array}{l} \{method()\} \\ \{attribute\} \end{array} \right\};$ 
```

---

*classname*

Name of the Java class whose method *method()* is to be executed, or whose attribute *attribute* is to be accessed

*method()*

Name of the method to be executed

*attribute*

Name of the attribute to be accessed

*Example*

```

<wtOnCreateScript>
<!--
// Generating a Java String
    myString = new WT_JAVA.java.lang.String("Hello world at " + new Date() +
                                             "\r\n");

// Calling a Java Method that converts the String into a ByteArray
    byteArray = myString.getBytes();

// Generating Java FileOutputStream (opening the file to be appended)
    fileout = new WT_JAVA.java.io.FileOutputStream("D:\\temp\\jtest.txt",
                                                true);

// Calling Java Method; Parameter is a JavaByteArray
    fileout.write (byteArray);

// Closing file
    fileout.close();

//-->
</wtOnCreateScript>

```

**12.1.6 Accessing class elements**

Class elements, i.e. attributes and methods, are accessed in WTScrip as follows.

---

```
var foo = WT_JAVA.classname. $\left. \begin{array}{l} \{method()\} \\ \{attribute\} \end{array} \right\};$ 
```

---

*classname*

Name of the Java class whose method *method()* is to be executed or whose attribute *attribute* is to be accessed

*method()*

Name of the method to be executed

*attribute*

Name of the attribute to be accessed

## 12.1.7 Invoking Java methods in WTSript

When invoking Java methods in WTSript, certain special features apply for return values, the transfer of parameters, and exception handling.

### Return values

In WTSript, the return values of Java methods are handled as follows:

- If the return value is defined with a primitive data type in the Java method signature, it can be used in WTSript as a value of that primitive data type.
- If the return value is a Java object, it is converted in the same way as a WTSript object, provided this is possible.
- If a Java method returns the value `null`, the WTSript result is `undefined`.

### Transferring parameters

When invoking Java methods, the following can be used as arguments:

- values of a primitive data type
- objects generated as a result of Java method calls

Other objects are not permitted as arguments when invoking Java methods.

Java objects of type `Byte`, `Short`, etc. are handled by Java as objects. Any conversions to primitive objects must be carried out explicitly.

The permitted parameter types and the type conversions carried out during the transfer of parameters are listed in the table on the next page.



WTSript type	Conversion to Java type
boolean/Boolean objects	Boolean
String objects	Default: <code>java.lang.String</code>  If the string consists of only one character, an attempt will also be made to convert this to type <code>char</code> .
number/Number objects	Default: <code>double</code>  If the object cannot be converted to the default type, an attempt will be made to convert it to all other numeric types until a suitable conversion is found, provided the conversion does not result in a loss of accuracy.
Java objects	Default: Type used when creating the object  If the object cannot be converted to the default type, an attempt will be made to convert it to all superclasses and implemented interfaces until a suitable conversion is found.
Java primitives	Default: Corresponding primitive Java type  If the object cannot be converted to the default type, an attempt will be made to convert it to all types for which the object value is valid until a suitable conversion is found. <code>char</code> values are not handled as numeric values.
Other WTSript objects	Not permitted as arguments

Type conversions in Java method calls and assignments

### *Executing a Java method call in WTScript*

When executing a Java method call  $m(arg-1, arg-2, \dots, arg-n)$ , WTScript proceeds as follows:

1. It searches the current object for a Java method  $m$  whose signature matches the default Java types:
  - If successful, the method  $m$  is executed.
  - If unsuccessful, WTScript repeats step 2 for each of the arguments  $arg-1, arg-2, \dots$
2. It searches for a method  $m$  whose signature is compatible with the relevant argument, i.e. whose corresponding formal parameter identifies a data type into which the argument can be converted:
  - If unsuccessful, a corresponding error message is output, and the method  $m$  is not executed.
  - If successful, WTScript repeats step 2 with the next parameter until all arguments have been checked.

If WTScript succeeds in finding a Java method  $m$  whose signature is compatible with  $arg-1$  through  $arg-n$ , then this method is executed.

### **Exception handling**

Any exceptions thrown when invoking a method are “converted” into WTScript exceptions and can be caught using the WTScript `try/catch` mechanism (see [section “Exception handling” on page 302](#)).

## **12.1.8 Reading and modifying attributes**

The rules that apply when invoking methods also apply to the reading of attributes. WTScript objects of type `Number`, `Boolean`, and `String` are converted to the primitive types `number`, `boolean`, and `string`. A check is then carried out to determine whether or not they are compatible with the attribute type.

If an attribute name cannot be found, this error is suppressed and no attribute is created. If an attempt is made to read a non-existent attribute, the result will be `undefined`.

## 12.1.9 Creating and using Java arrays in WTSript

Java objects are created in WTSript using the `new` operator. The elements of the array are not initialized.

---

```
var = new WT_JAVA.classname [size];
```

---

*classname*

Name of the class to which the array elements belong. This must be a fully qualified class name.

*size*

Number of elements to be created, specified in numeric or string format (e.g. "12"). Variables may also be used here.

Strings in the format `[0-9]*[0-9a-zA-Z]*` will be accepted, whereby any digits at the beginning of the string will be converted to numeric format, while the rest of the string is ignored. If the string does not begin with a digit, the specification will be invalid and will result in an error.

### Arrays containing elements of a primitive Java data type

Arrays containing elements of a primitive Java data type cannot be created directly in WTSript using the `new` operator, but they can be created using the following class:

```
public class ArrayFactory {
    private ArrayFactory() {
    }
    public static byte[] newByteArray(int i) {return new byte[i];}
    public static short[] newShortArray(int i) {return new short[i];}
    public static int[] newIntArray(int i) {return new int[i];}
    public static long[] newLongArray(int i) {return new long[i];}
    public static float[] newFloatArray(int i) {return new float[i];}
    public static double[] newDoubleArray(int i) {return new double[i];}
    public static char[] newCharArray(int i) {return new char[i];}
    public static boolean[] newBooleanArray(int i) {return new boolean[i];}
}
```

### Multidimensional arrays

To create multidimensional arrays, you must use the interfaces of `java.lang.reflect.Array`.

*Example:*

```
m = new WT_JAVA.MyObject; ----- (1)
c = m.getClass();
MyObjectArray = WT_JAVA.java.lang.reflect.Array.newInstance(c,10); (2)
                                     //Array of MyObjects, 10 elements
c = MyObjectArray.getClass(); ----- (3)
MultiArray = WT_JAVA.java.lang.reflect.Array.newInstance(c, 20); ----- (4)
```

Statements (1) - (4) correspond to the following Java statement:

```
MyObject[][] MultiArray = new MyObject[20][10];
```

Statements (1) - (2) can be replaced by the following statement:

```
MyObjectArray = new WT_JAVA.MyObject[10];
```

It is possible to access individual array elements using the usual WTSyntax syntax. When assigning values to array elements, the same rules apply as for fields. In this case, however, the transferred index is checked for validity before access takes place.

Assignments between array variables are subject to the type checking rules defined in Java. Since the Java implementation of WTSyntax checks all types before assignment, it takes care of error handling. If certain statements are found to violate the type rules, a Java exception will not be thrown.

WTSyntax arrays are never converted into Java arrays. They are handled in the same way as other WTSyntax objects and are ignored during interaction with Java.

### 12.1.10 Using WTSyntax operators with Java objects

WTSyntax operators can also be applied to Java objects. When using the “+” operator, Java objects are always converted to string format.

If you wish to perform numeric calculations, all objects must first be converted to numeric format.

*Example:*

Instead of

```
x = o1 + o2;           // String operation
```

use

```
x = 1*o1 + 1*o2;      // Addition
```

### 12.1.11 Example

The following example creates a Java string, converts it to a byte array, and outputs it to a file.

```
<wtOnCreateScript>
<!--
// Create a Java string
myString = new WT_JAVA.java.lang.String("Hello world at " + new Date() + "\r\n");
// Invoke Java method for converting the string into a byte array
byteArray = myString.getBytes();
// Create Java FileOutputStream (open file for writing)
fileout = new WT_JAVA.java.io.FileOutputStream("D:\\temp\\jtest.txt", true);
// Invoke Java method and transfer Java byte array as an argument
fileout.write (byteArray);
// Close the file
fileout.close();
//-->
</wtOnCreateScript>
```

## 12.2 Using C/C++ user exits

C/C++ user exits are called by means of the methods of the `WT_Userexit` class (see [section “WT\\_Userexit class” on page 258](#)).

You can also use a number of different user exit libraries.

### 12.2.1 Files supplied for supporting C/C++ user exits

To support user exits, WebTransactions provides the following files in the directory `install_dir/lib`:

`WTUserexit.c`

Sources for user exits. A number of examples are already present and you can add your own user exits.

`WTPublic.h`

Header file containing the functions which can be implemented in a user exit and which map the WebTransactions statements and evaluation operator.

`WTUserexits.so` (for Unix platforms) / `WTUserexits.dll` (for Windows)

Shared library in which the shipped example user exits are already included. WebTransactions uses this library as the default library if no library is specified. You can link your own user exits (see below). However, in this case you should copy the library from the installation directory to the base directory. Alternatively, you can generate your own user exit library.

`WTSystemExits.so` (for Unix platforms) / `WTSystemExits.dll` (for Windows)

Shared library which contains the user exits used by the start templates, e.g. `Getfile` (see [“Example 1” on page 353](#)).

These ready-made user exits are described in [section “Ready-made C/C++ user exits supplied with WebTransactions” on page 355](#). In the case of BS2000/OSD, they are contained in the `WTHolder` program.

## 12.2.2 Defining C/C++ user exits

You define the user exit function as:

---

```
char *NewUserExit (void *holder, int ac, char *av[]);
```

---

<i>holder</i>	Control structure for the session
<i>ac</i>	Contains the number of arguments
<i>av</i>	Contains the arguments specified when the <code>WT_Userexit.function(...)</code> method is called.

The function must return a string which is read by the template.



The WebTransactions working directory is the session directory under *tmp*. If the user exit initiates a process whose lifetime looks set to extend beyond the end of the current WebTransactions session, you must change the working directory for this process. Otherwise, WebTransactions will not be able to delete the temporary session directory at the end of the session.

## 12.2.3 Linking C/C++ user exits

The way you link user exits differs depending on the WebTransactions platform.

On Unix platforms and Windows platforms, you either integrate the user exits in the shared default library `WTUserexits.{so|dll}` or make them available in your own shared libraries.

Under OSD, the user exits are integrated in the WTHolder program.

### Windows

To compile the user exits you have defined and include these in the shared library `WTUserexits.dll`, use a development environment which enables you to generate dll libraries, e.g. Visual C++.

In the development environment, create a project that generates a dll library. You can then add the sample source code `WTUserexit.c` to this project.

Since user exits are called using the name of the dll library, it is possible to create various dll libraries with different functionalities.

If you use WebTransactions functions from `WTPublic.h` in user exits, you must also include the kernel `WTKernel.lib`.

### Unix platforms (only WebTransactions supply unit openUTM/OSD/MVS)

To compile the user exits which you have defined and include these in the shared library *library*, enter the following call:

---

```
cc -G -share -o library.so userexit1.c . . . userexitn.c
```

---

*library.so*

*library* specifies the library. If no library *library.so* exists as yet then one is generated.

*userexit1.c . . . userexitn.c*

Sources for C/C++ functions which are to be compiled and included

### OSD (only WebTransactions supply unit openUTM/OSD)

Under POSIX, you cannot use any shared libraries: all user exits must be statically linked to the WTHolder program. To do this, remake the WTHolder program - to include the user exits - using the shipped make file. All the WTHolder program modules are available to you in the following library:

*install\_dir/lib/libWTHolderUTMV4.a* (for WTHolderUTMV4)

Proceed as follows:

- Add your new user exits to the shipped file *WTuserexits.c*.
- Check whether the name of your user exit is the same as the name of a shipped user exit. Since no user exit may possess the same name as the user exits shipped with WebTransactions, you will have to rename your user exits if the names correspond. This is particularly important if you develop exits supplied in the source code, such as *Getfile*, to create your own user exits.
- Use the shipped make file (*install\_dir/lib/Makefile*) to generate a new program *WTHolderUTMV4*.
- Copy this new program to the base directory under the name *WTHolder*.



You may need to adapt the names used in the make file for the UPIC and CMX libraries to your current system.



## 12.2.4 Examples of C/C++ user exits

### *Example 1*

The following user exit returns the contents of a text file. It can be found in the shipped user exit library `WTSYSTEMEXITS.{dll|so}`.

```

/*****
/*
/* Getfile( file )
/*
/*
/*****
/*
/* The contents of the textfile file is returned.
/*
/*
/* This exit is used by predefined start pages (wtstart.htm, wtstartUTM.htm,..)
/* and should neither be modified nor removed !!!
/*
/*
/*****
char* Getfile(void *wtholder, int ac, char *av[])
{
    FILE*      p_file;
    int        bytesRead;
    char*      ct;

    if ( ac == 0 )
        return( "" );

    p_file = fopen( av[0], "r" );    /* open file for reading */
    if ( p_file == NULL )
        return( "" );

    /* Since the caller does not free the returned Data reuse the buffer */
    returnStringLength = 0;
    if ( returnStringSize - returnStringLength < 2 )
        returnString = realloc( returnString, returnStringSize += 1024 );

    /* read the Lines in the file and append them to the return string */
    while( ( bytesRead = fread( returnString + returnStringLength, 1,
                                returnStringSize - returnStringLength, p_file ) ) != 0 )
    {
        returnStringLength += bytesRead;
        if ( returnStringSize - returnStringLength < 2 )
            returnString = realloc( returnString, returnStringSize += 1024 );
    }
    returnString[returnStringLength] = '\0';
    return returnString;
}

```

*Example 2*

This example illustrates the user exit `UXEurope` which is used to evaluate the image coordinates of a “clickable image”:

```
char *UXEurope (wt_holderCommId *wtholder, int ac, char *av[])
{
    int x,y;

    if (ac == 2)
    {
        x = (int) atoi (av[0]);
        y = (int) atoi (av[1]);

        if (x > 180 && y > 90 && x < 270 && y < 125)
            return ("1"); /* Belgium */
        if (x > 160 && y > 184 && x < 228 && y < 213)
            return ("2"); /* France */
        if (x > 232 && y > 60 && x < 325 && y < 90)
            return ("3"); /* Germany */
        if (x > 400 && y > 260 && x < 500 && y < 310)
            return ("4"); /* Greece */
        if (x > 280 && y > 210 && x < 350 && y < 260)
            return ("5"); /* Italy */
        if (x > 0 && y > 240 && x < 100 && y < 280)
            return ("6"); /* Portugal */
        if (x > 40 && y > 290 && x < 120 && y < 330)
            return ("7"); /* Spain */
        if (x > 200 && y > 140 && x < 330 && y < 180)
            return ("8"); /* Switzerland */
        if (x > 70 && y > 20 && x < 250 && y < 60)
            return ("9"); /* United Kingdom */
    }
    return ("0");
}
/wtOnCreateScript>
```

## 12.3 Ready-made C/C++ user exits supplied with WebTransactions

WebTransactions comes with a set of ready-defined C/C++ user exits, some of which are used internally by WebTransactions, e.g. in the start templates.

The user exits described in this section, however, can also be used in templates which you have created or customized yourself. The source code of these user exits is provided in the file `WTUserExits.c` for your own use (with just one exception: the source code for the `GetInstallDir` user exit is **not** supplied).

The ready-defined user exits can be found in different locations, depending on the WebTransactions platform:

**OSD**      In `WTHolder`; these user exits must be addressed as if they were in a library named `WTSYSTEMEXITS`

**Unix platform**  
In the library `WTSYSTEMEXITS.SO`

**Windows**   In the library `WTSYSTEMEXITS.DLL`

The following table provides an overview of these user exits:

Name	Function	Described in
<code>CheckLogin</code>	Checks whether or not the specified password is assigned to the user name.	Section <a href="#">“Check-Login”</a> on page 357
<code>CheckProcess</code>	Checks whether or not the specified process ID exists in the system and whether or not the process is a <code>WTHolder</code> process.	Section <a href="#">“Check-Process”</a> on page 357
<code>Creationtime</code>	Returns the creation time of the specified file in <code>cfile</code> format.	Section <a href="#">“Creationtime”</a> on page 358
<code>Delfile</code>	Deletes a file. This user exit can only be executed within an administration process.	Section <a href="#">“Delfile”</a> on page 359
<code>FreeBuffer</code>	Releases the buffer shared by <code>GetDir</code> and <code>GetFile</code> .	Section <a href="#">“FreeBuffer”</a> on page 359
<code>FreeNameInPool</code>	Releases the first entry in a list reserved by the process (default). This user exit can also be used to release a particular name from the list or all reserved entries.	Section <a href="#">“FreeNameInPool”</a> on page 359
<code>Getdate</code>	Returns the specified time in <code>cfile</code> format.	Section <a href="#">“Getdate”</a> on page 360

Name	Function	Described in
Getdir	Returns the names of all files that are located in the specified directory and (optionally) match a specified pattern.	Section <a href="#">“Getdir” on page 360</a>
Getfile	Returns the contents of the specified file in string format.	Section <a href="#">“Getfile” on page 360</a>
GetInstallDir	Returns the WebTransactions installation directory. The source code of this user exit is <b>not</b> supplied.	Section <a href="#">“GetInstallDir” on page 361</a>
Gettime	Returns the current time in cfile format.	Section <a href="#">“Gettime” on page 361</a>
LockNameInPool	Reserves the first free entry in a list. This user exit can also be used to reserve a particular name in the list.	Section <a href="#">“LockNameInPool” on page 362</a>
Modificationtime	Returns the time at which the specified file was last modified.	Section <a href="#">“Modificationtime” on page 362</a>
Putfile	Generates a file containing the transferred character string. This user exit can only be executed within an administration process.	Section <a href="#">“Putfile” on page 363</a>
ReleaseStationName	Releases a name reserved with ReserveStationName.	Section <a href="#">“ReleaseStationName” on page 363</a>
ReplaceByConfigFile	Replaces one character string with another under the control of a replacement table.	Section <a href="#">“ReplaceByConfigFile” on page 364</a>
ReserveStationName	Ensures that all names are unique, e.g. checks whether or not the specified station name for a connection to an OSD application has already been used.	Section <a href="#">“ReserveStationName” on page 364</a>
SendMail	Sends a message with the SMTP protocol (text only, no MIME support).	Section <a href="#">“SendMail” on page 365</a>
WTSleep	places the holder on waittime for a set number of milliseconds.	Section <a href="#">“WTSleep” on page 366</a>

### 12.3.1 CheckLogin

---

```
CheckLogin( config_file, user_name, password )
```

---

**Function:**

Checks whether or not the specified password is assigned to the user name. *config\_file* is sought relative to the base directory, and must consist of two columns separated by white space characters (see [page 30](#)).

**Result:**

If the password is assigned to the user name, the user name is returned. Otherwise, the user exit returns an empty string.

*Example:*

Contents of *basedir/config\_file*:

```
user      password
smith     key
...
```

### 12.3.2 CheckProcess

---

```
CheckProcess( process_id )
```

---

**Function:**

Checks whether or not the specified process ID exists in the system and whether or not the process is a WTHolder process.

**Result:**

"alive"

The specified process ID exists in the system and the process is a WTHolder process.

"dead" or "" (empty string)

The specified process ID does not exist in the system or the process is not a WTHolder process.

### 12.3.3 Creationtime

---

Creationtime( *file\_name* )

---

**Function:**

Returns the creation time of the specified file in ctime format. The path can be given as an absolute or relative specification. Relative path specifications refer to the temporary session directory.

**Result:**

The creation time in string format. If the file does not exist, a question mark is returned.

*Example*

```
<wtoncreatescript>
<!--
ex = new WT_Userexit('WTSYSTEMExits');
document.writeln('<br> Creationtime("session
info")=',ex.Creationtime('../'+WT_SYSTEM.SESSION+'.info'));
document.writeln('<br>
Creationtime(WT_SYSTEM.BASEDIR+"/basedir_file")=',ex.Creationtime(WT_SYSTEM.B
ASEDIR+'/config'));
document.writeln('<br>
Creationtime("c:/abs_file")=',ex.Creationtime('c:/windows'));
document.writeln('<br>
Creationtime("unknown_file")=',ex.Creationtime('unknown_file'));
//-->
</wtoncreatescript>
```

The example generates the following output:

```
Creationtime("session info")=Tue Jun 08 16:10:28 2010
Creationtime(WT_SYSTEM.BASEDIR+"/basedir_file")=Tue Jun 08 12:37:23 2010
Creationtime("c:/abs_file")=Thu Nov 02 13:18:34 2006
Creationtime("unknown_file")=?
```

### 12.3.4 Delfile

---

```
Delfile( file_name )
```

---

Function:

Deletes the specified file. The path can be given as an absolute or relative specification. Relative path specifications refer to the temporary session directory.

Result:

Empty string if the user exit is successful; otherwise, an error message.

### 12.3.5 FreeBuffer

---

```
FreeBuffer()
```

---

Function:

Releases the buffer shared by `GetDir` and `GetFile`.

Result:

No return value.

### 12.3.6 FreeNameInPool

---

```
FreeNameInPool ( config_file [, {name_if_not_first_reserved_by_this_process | "ALL"} ] )
```

---

Function:

Releases the first entry in a list reserved by the process (default). This user exit can also be used to release a particular name from the list or all reserved entries.

Result:

Name of the released entry or an empty string. An empty string is returned if all entries were released or if it was not possible to release any entries.

See also:

[LockNameInPool](#)

*Example:*

See `LockNameInPool`.

### 12.3.7 Getdate

---

```
Getdate( numeric_time_value )
```

---

Function:

Returns the specified time in ctime format.

Result:

Time in string format or an empty string if the time value was invalid.

### 12.3.8 Getdir

---

```
Getdir( dir_name_relative_to_basedir [ , pattern ] )
```

---

Function:

Returns the names of all files that are located in the specified directory and (optionally) match a specified pattern.

Result:

String containing the file names separated by new-line characters.



After the last `Getdir/Getfile` call, you should use `FreeBuffer()` to release the buffer space used.

### 12.3.9 Getfile

---

```
Getfile( file_name )
```

---

Function:

Returns the contents of the specified file. Relative path specifications refer to the temporary session directory.

Result:

Contents of the file in string format.



After the last `Getdir/Getfile` call, you should use `FreeBuffer()` to release the buffer space used.



### 12.3.10 GetInstallDir

---

GetInstallDir()

---

Function:

Returns the WebTransactions installation directory.

Result:

Installation directory.

The source code of this user exit is **not** supplied.

### 12.3.11 Gettime

---

Gettime()

---

Function:

Returns the current time in ctime format.

Result:

Time in string format.

### 12.3.12 LockNameInPool

---

```
LockNameInPool( config_file [, name] )
```

---

**Function:**

Reserves the first free entry in a list. This user exit can also be used to reserve a particular name in the list. If a name is marked as reserved but the process that reserved it no longer exists, it is treated as free and reassigned.

**Result:**

Name of the reserved entry or an empty string.

**See also:**

[FreeNameInPool](#)

*Example:*

Contents of *basedir/config\_file*:

```
FREE          STATION1  comment
FREE          STATION2  comment
...
```

The first 16 bytes of *config\_file* contain "FREE " or the process ID of the reserving process. This is followed by a name and a comment, usually separated by white space characters (see [page 30](#)).

### 12.3.13 Modificationtime

---

```
Modificationtime( file_name [, 'N' ])
```

---

**Function:**

Returns the time at which the specified file was last modified.

**Result:**

String in ctime format or - if the optional second argument 'N' was specified in the call - in numeric format (internal time format).

### 12.3.14 Putfile

---

```
Putfile( file_name, content, length )
```

---

**Function:**

Generates a file containing the transferred string. Relative path specifications refer to the temporary session directory.

**Result:**

Empty string if the user exit is successful; otherwise, an error message.

### 12.3.15 ReleaseStationName

---

```
ReleaseStationName ( station_name )
```

---

**Function:**

Releases a name reserved with `ReserveStationName`.

**Result:**

OK

The station name was found and the corresponding entry was deleted from the file containing used station names.

NOTOK

The value specified in *station\_name* is not a valid station name.

ERROR

An error occurred while executing the user exit. In this case, you can assume that the specified station name has not been released.

**See also:**

[ReserveStationName](#)

### 12.3.16 ReplaceByConfigFile

---

```
ReplaceByConfigFile( replace_file , key_to_be_replaced )
```

---

**Function:**

Replaces one character string with another under the control of a replacement table in *replace\_file*. *replace\_file* must consist of two columns separated by white space characters (see [page 30](#)).

**Result:**

Replacement string if the entry was found; otherwise, an empty string.

*Example:*

Contents of *basedir/replace\_file*:

```
key1          replacement1    COMMENT!
123.45.6.7    STATION2      relation ip-address to station name
123.45.6.8    USERx           relation ip-address to user name
user1         station1    relation user name to station
...
```

### 12.3.17 ReserveStationName

---

```
ReserveStationName( station_name )
```

---

**Function:**

Ensures that all names are unique, e.g. checks whether or not the specified station name for a connection to an OSD application has already been used.

**Result:**

OK

The station name is currently not in use. It is reserved and a corresponding entry is generated in the file containing used station names.

NOTOK

The specified station name is currently in use or is not defined.

ERROR

An error occurred while executing the user exit. In this case, you can assume that the specified station name cannot be used.

**See also:**

[ReleaseStationName](#)

## 12.3.18 SendMail

---

SendMail ( *Server, From, To, CC, BCC, Subject, Body, Header* )

---

### Function:

Sends a message with the SMTP protocol to one or more recipients. You must set the following parameters:

*Server* Internet address or symbolic name of the mail server

*From* Mail address of the sender

*To* Mail address of the recipient. You can also specify multiple mail addresses separated by a semi-colon “;”

*CC* Abbreviation for “Carbon Copy”; mail address of the recipient of the copy. You can also specify multiple mail addresses separated by a semi-colon “;”

*BCC* Abbreviation for “Blind Carbon Copy”; mail address of the recipient of the copy. You can also specify multiple mail addresses separated by a semi-colon “;”

*Subject* \*Subject line of the mail.

*Body* Text of the mail.

*Header* Header line for the mail. The string must contain any necessary linefeeds.

*Server, From, To, Subject* and *Body* are mandatory parameters. The *CC* and *BCC* parameters can also be passed as empty strings.

### Result:

The `SendMail` method returns the string `OK` or an error message if it was not possible to establish the connection. If an error occurs during communication with the SMTP server then this is logged in the `WebTransactions` trace file.

The following return values are possible:

“OK”

“Incomplete function call.”

“Memory allocation failed.”

“Function call: `WSAStartup()` failed.”

“Creation of a socket failed.”

“Function call: `gethostbyname()` failed.”

“Connection to socket failed.”

“Receive from socket failed.”

“Send to socket failed.”

*Example:*

```
<wtoncreatescript>
<!--
var MailServer = "smtpmail.server.de";
var MailFrom = "Bundestrainer@dfb.de";
var MailTo1 = "Terrier <Berti.Vogts@unknown.de>";
var MailTo2 = "Franzl <Franz.Beckenbauer@fcb.com>";
var MailCc = "Papst@vatican.va";
var MailBcc = "";
var MailSubject = "WM2010";
var MailBody = "Der Ball ist rund und das Spiel dauert 90 Minuten.";
var MailHeader = "Content-Type: text/plain; charset=ISO-8859-1";
SMTPExit = new WT_Userexit();
SMTPExit.SendMail(MailServer,MailFrom,MailTo1+';'+MailTo2,MailCc,
MailBcc,MailSubject,MailBody);
delete SMTPExit;
//-->
</wtoncreatescript>
```

## 12.3.19 WTSleep

---

WTSleep( [*waittime*] )

---

**Function:**

This system exit places the holder on *waittime* for a set number of milliseconds.

*waittime*

Expression that is converted into the type `number` and specifies the number of milliseconds of wait time left. If no parameter or an invalid parameter is specified the default value 1000 ms is used.

**Note:** the wait on the OSD platform always lasts at least 1 second.

**Result:**

The string `OK` is returned.

---

## 13 XML conversion

This chapter describes the basic principles of XML conversion for:

- the portable representation of data for communication with external applications via XML messages (XML=eXtended Markup Language) in [section “Importing and exporting XML documents” on page 367](#)
- the conversion of WTScrip data structures to XML documents and vice versa in [section “Exporting data structures” on page 372](#)

For information on how to use the `WT_Filter` class for communication between WebTransactions applications (conversion of WTML function calls), please refer to the WebTransactions manual “Client APIs for WebTransactions”.

### 13.1 Importing and exporting XML documents

This section describes how to convert XML documents into WTScrip data structures and vice versa. This allows for communication with any external application that creates or processes XML documents.

If XML documents are imported using the `WT_Filter` class, they must be transformed into an internal representation of a WTScrip object tree. For this purpose, the structure of the XML document is mapped to an object tree, each of whose leaves represents an XML element. The following sections describe the format of these data objects.

If you wish to export WTScrip data structures using the `objectTreeToXML` method, these data structures must be converted into the XML object tree format described below before they can be exported.

### 13.1.1 Structure of an imported XML object

In WebTransactions, XML elements are represented as follows:

---

```
name
attribute
child
0
1
...
```

---

name

This attribute is of data type `string` and specifies the name of the XML element.

attribute

This attribute is an object of type `object` from the `Object` class which contains an attribute for each attribute of the XML element.

child

This attribute is an object of type `object` from the `Object` class. If the current XML element contains subordinate elements, this object contains an attribute for each subelement type with the name of the XML element type. References to the objects 0, 1, ... of the respective type are inserted as attributes of this subobject (see description below).

0, 1, ...

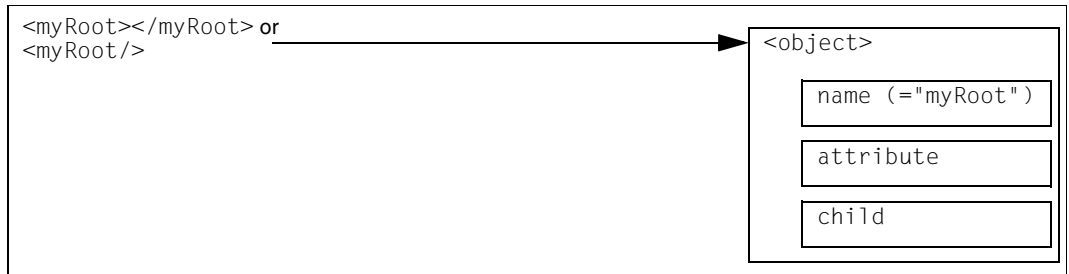
For each XML element within the current XML element, an attribute is created with the index of the XML subelement as the name, the data type `object`, and the class `Object` which is in turn structured as an XML object.



### 13.1.2 Representation of XML elements

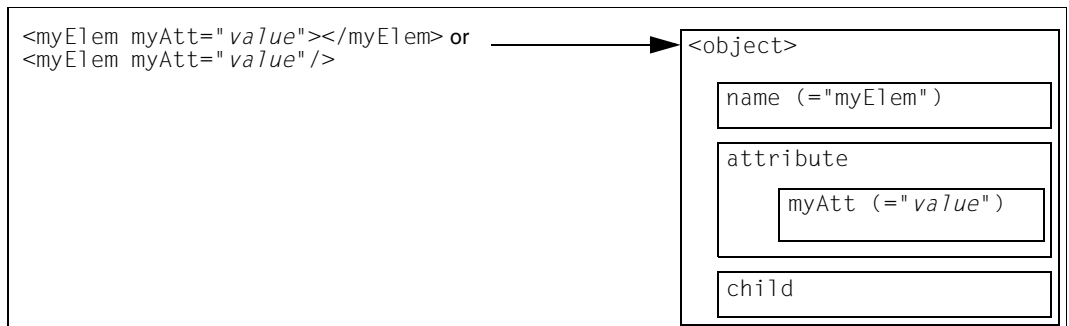
The following figures explain the elements of WTSript data structures described above.

#### Representation of a simple XML element



The XML element `myRoot` is mapped to an object in which the `name` attribute is set to the name of the XML element (`myRoot`).

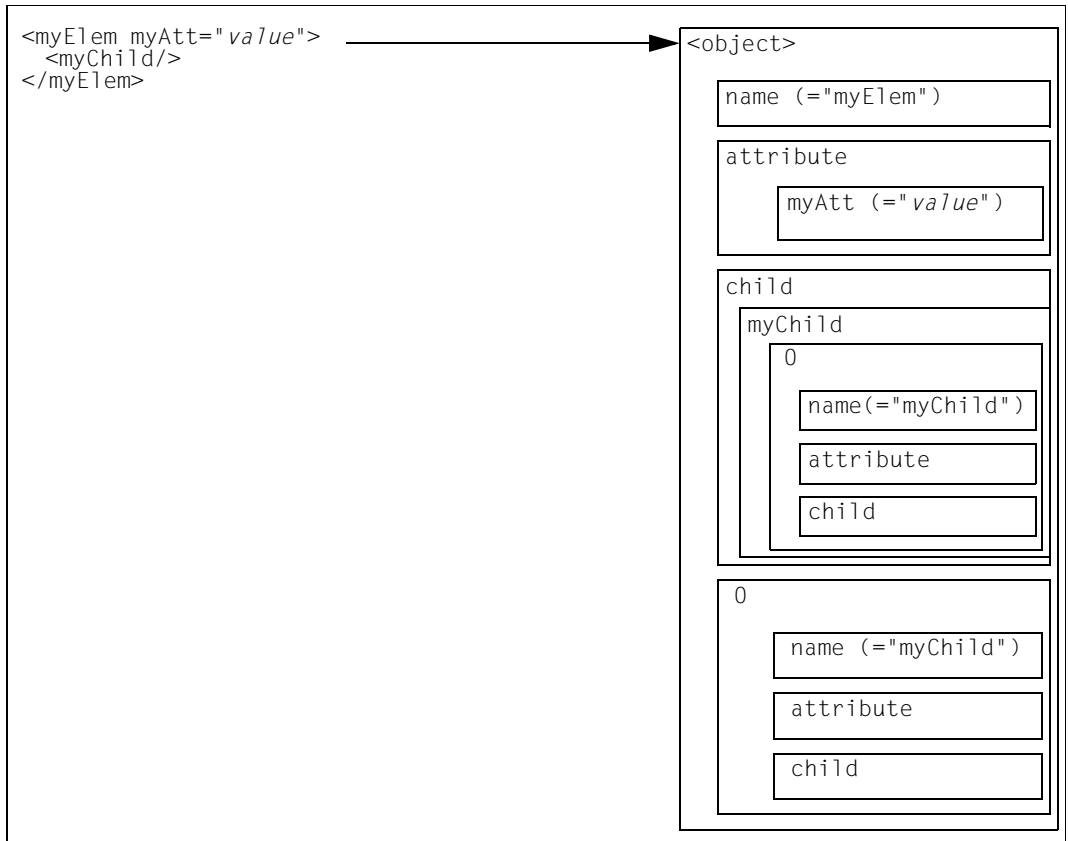
#### Representation of an XML element with attributes



The XML element `myElem` is mapped to an object in which the `name` attribute is set to the name of the XML element (`myElem`).

For each attribute of `myElem`, the object's `attribute` attribute is assigned a separate attribute with the name of the XML attribute (in our example, `myAtt`) and the value of the XML attribute (in our example, `value`).

## Representation of subelements of an XML element



The XML element `myElem` is mapped to an object in which the `name` attribute is set to the name of the XML element (`myElem`).

For each attribute of `myElem`, the object's `attribute` attribute is assigned a separate attribute with the name of the XML attribute (in our example, `myAtt`) and the value of the XML attribute (in our example, `value`).

In addition, for each subelement of `myElem`, a separate subobject is created in `<object>` which is named in accordance with the index of the subelement (0, 1, 2, 3, ...) and is structured recursively in this way (in our example, the object 0).

Furthermore, for each XML element type (here, `myChild`), the `child` attribute is assigned an object which in turn comprises references to the subobjects (0, 1, ...) belonging to this object type (see also the following diagram).

## Representation of several subelements

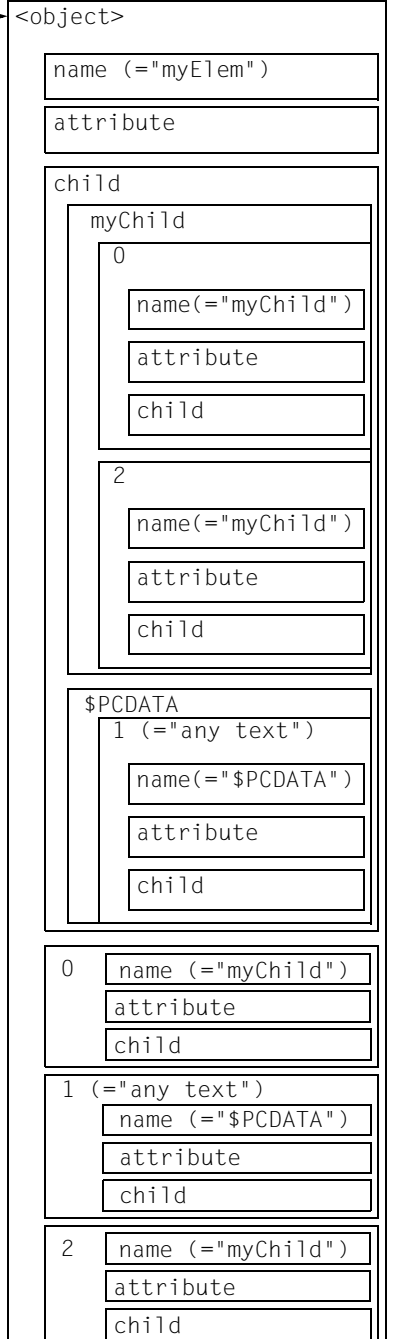
```
<myElem>
  <myChild>
    any text
  </myChild>
</myElem>
```

If there are several subelements, these are created as described on the previous page:

Each subelement is assigned an object whose name corresponds to the index of the subelement in the element (here 0 for 1st `myChild`, 1 for text, 2 for 2nd `myChild`).

For each element type (here `myChild` and `$PCDATA` for standard text), the `child` attribute is assigned a subobject which contains references to the subobjects 0, 1, 2, ... of the object, corresponding to the respective element type.

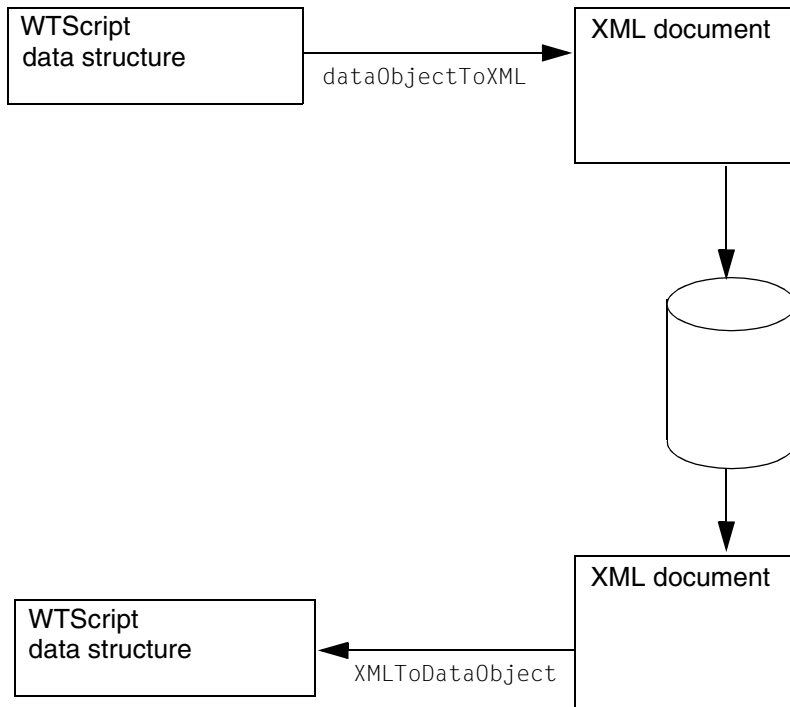
Standard text is always stored under the name `$PCDATA`. A special feature here is that the text is saved as a value of the subobject.



## 13.2 Exporting data structures

This section describes how WTSript data structures are exported and imported (using the methods `XMLToDataObject` and `dataObjectToXML`). This means that WTSript data structures can be transferred in a saveable format, then stored and reloaded (using the methods of the `document` class), and subsequently reconverted into the corresponding WTSript data structures.

The data structures are converted using the two methods `XMLToDataObject` and `dataObjectToXML`, whereby no particular conventions need be observed. For the conversion, WebTransactions uses the following DTD (Document Type Definition, a description of how an XML document is structured), which is provided here for information purposes.



## DTD for representing WTScrip data structures in XML

The following DTD applies to the representation of WTScrip data structures as XML documents:

```

<!ELEMENT data      ((undefined|number|boolean|
                      string|object)*)>
<!ELEMENT undefined EMPTY>
<!ELEMENT number   (#PCDATA)>
<!ELEMENT boolean  (#PCDATA)>
<!ELEMENT string   (#PCDATA)>
<!ELEMENT object   (#PCDATA?
                    (undefined|number|boolean|
                     string|object|function)*)>
<!ELEMENT function EMPTY>
<!ATTLIST undefined name          CDATA   #REQUIRED>
<!ATTLIST number   name          CDATA   #REQUIRED>
<!ATTLIST boolean  name          CDATA   #REQUIRED>
<!ATTLIST string   name          CDATA   #REQUIRED>
<!ATTLIST object   name          CDATA   #REQUIRED
                    class        CDATA   #IMPLIED
                    reference     CDATA   #IMPLIED>
<!ATTLIST function name          CDATA   #REQUIRED>

```

As well as its use within WebTransactions, exporting the data structures according to this DTD also allows external applications to access WTScrip data structures saved in this way.



---

## 14 Examples

The examples in this chapter illustrate the ways in which the different WTML tags interact.

### 14.1 Changing styles

This example illustrates how you add a button to change between styles and handle the return value in WebTransactions. In this case, the `STYLE` button is polled in the `If` condition:

- If it is pressed then the `STYLE` system attribute is reset. Since in this case there is no communication with the host application, the `FORMAT` system attribute remains unchanged. Consequently, a template is now read which corresponds to the current template but possesses a different interface style. WebTransactions does not now search in the standard template directory `config/forms` but in `config/graphic`.
- If the `STYLE` button is not pressed, communication with the host is performed. The template for the next dialog step is therefore displayed and the interface style remains unchanged.

*HTML template:*

```
...
<Input Type="SUBMIT" Name="STYLE" Value="Graphic">
...
<wtOnReceiveScript>
if ( WT_POSTED.STYLE == "Graphic")
    WT_SYSTEM.STYLE ="graphic";
else
{
    WT_HOST.std.send();
    WT_HOST.std.receive();
}
</wtOnReceiveScript>
```

## 14.2 Polling the Exit button

To permit users to exit the application in a structured way, the `Exit` button is polled in the `If` condition. If it is pressed, a final message is output and the session is terminated with the `exitSession()` global function. Otherwise communication with the host is performed

*HTML template:*

```
....
<input Type="SUBMIT" Name="Exit" Value="End">
<wtRem End or communicate with host>
<wtOnReceiveScript>
if ( WT_POSTED.Exit == "End" )
  {
document.write("End of session...");
  WT_HOST.std.close();
  exitSession();
  }
else
  {
  WT_HOST.std.send();
  WT_HOST.std.receive();
  }
</wtOnReceiveScript>
```



## 14.3 Saving data with XML conversion

This example is based on the assumption that a WebTransactions application is to manage user-specific data (display style, which can be selected by the user, as well as various user data such as the user number, last usage of the application, etc.).

The user identifies himself or herself using a number, which also serves as the basis for storing the data.

The user-specific data is stored in the following data structure:

```
function UserData() { // Constructor
    // Class attributes:
    this.objectName = "";           // Name of data object
    this.userNumber = 0;
    this.style = "";
    this.lastUsage = new Date;
    // Methods (saving and loading of data)
    this.save = saveUserData;
    this.load = loadUserData;
}
```

The two methods `saveUserData` and `loadUserData` implement the operations for XML conversion and data storage:

```
function saveUserData() {
    XMLString = new String;

    // Save current style:
    this.style = WT_SYSTEM.STYLE
    // Convert data to XML:
    XMLString = WT_Filter.dataObjectToXML(this.objectName);
    // Write data to file <WT_SYSTEM.BASEDIR>/<userNumber>.wtd:
    WT_Userexit.Putfile(WT_SYSTEM.BASEDIR + "/" + this.userNumber + ".wtd",
        XMLString, XMLString.length);
}

function loadUserData() {
    XMLString = new String;

    // Load data from file <WT_SYSTEM.BASEDIR>/<userNumber>.wtd:
    XMLString = WT_Userexit.Getfile(WT_SYSTEM.BASEDIR + "/"
        + this.userNumber + ".wtd")
    // Convert XML to data (in object this.objectName):
    WT_Filter.XMLToDataObject(XMLString);
    // Restore saved style:
    WT_SYSTEM.STYLE = this.style;
}
```



---

## 15 Short reference guide

### 15.1 WTML tags

Function	Syntax
Comment tag	<code>&lt;wtRem <i>comments</i>&gt;</code> or: <code>&lt;wtRem&gt;</code> <i>comments</i> <code>&lt;/wtRem&gt;</code>
Dataform tag	<code>&lt;wtDataform [Name="<i>name</i>" ] [OnSubmit="<i>OnSubmitHandler</i>" ]</code> <code>[ASYNC_PAGE="<i>asyncPage</i>" ]&gt;</code> <i>Area</i> <code>&lt;/wtDataform&gt;</code>
Exit tag	<code>&lt;wtExit scope={"TEMPLATE"   "DIALOGSTEP"   "SESSION"}&gt;</code>
Include tag	<code>&lt;wtInclude Name="<i>fileName</i>"&gt;</code>
IF tag	<code>&lt;wtIf (<i>condition</i>)&gt;</code> <i>Block1</i> [ <code>&lt;wtElse&gt;</code> <i>Block2</i> ] <code>{&lt;wtEndIf&gt;   &lt;/wtIf&gt;}</code>
DO WHILE tag	<code>&lt;wtDoWhile (<i>condition</i>)&gt;</code> <i>Block</i> <code>&lt;/wtDoWhile&gt;</code>
DO UNTIL tag	<code>&lt;wtDo&gt;</code> <i>Block</i> <code>&lt;wtUntil(<i>condition</i>)&gt;</code>
OnCreateScript tag	<code>&lt;wtOnCreateScript&gt;</code> <i>CreateScript</i> <code>&lt;/wtOnCreateScript&gt;</code>
OnReceiveScript tag	<code>&lt;wtOnReceiveScript&gt;</code> <i>ReceiveScript</i> <code>&lt;/wtOnReceiveScript</code>

## 15.2 WTSript statements (alphabetic order)

Function	Syntax
Empty statement	;
Expression as statement	<i>expression</i> ;
Statement block as statement	{ <i>block</i> ...}
if branch	if ( <i>condition</i> ) <i>block1</i> [ else <i>block2</i> ]
while loop	[ <i>label</i> : ]while ( <i>condition</i> ) <i>block</i>
do/while loop	[ <i>label</i> : ]do <i>block</i> while ( <i>condition</i> );
for loop	[ <i>label</i> : ]for ( [ <i>init</i> ]; [ <i>condition</i> ]; [ <i>update</i> ] ) <i>block</i>
for/in loop	[ <i>label</i> : ]for ( <i>name</i> in <i>object</i> ) <i>block</i>
switch statement	switch ( <i>expression</i> ) { {case <i>label</i> : <i>block1</i> ... [break;] }... [default: <i>block2</i> ...] }
break statement	break [ <i>label</i> ];
continue statement	continue [ <i>label</i> ];
return statement	return [ <i>retValue</i> ] ;
Variable declaration and assignment	var { <i>identifier</i>   <i>identifier</i> = <i>value</i> } [ { , <i>identifier</i>   , <i>identifier</i> = <i>value</i> }... ] ;
function statement	function <i>identifier</i> ( [ <i>parameter</i> { , <i>parameter</i> }... ] ) { [ <i>block</i> ... ] }
with statement	with ( <i>object</i> ) <i>block</i>

---

# Glossary

A term in *->italic* font means that it is explained somewhere else in the glossary.

## active dialog

In the case of active dialogs, WebTransactions actively intervenes in the control of the dialog sequence, i.e. the next *->template* to be processed is determined by the template programming. You can use the *->WTML* language tools, for example, to combine multiple *->host formats* in a single *->HTML* page. In this case, when a host *->dialog step* is terminated, no output is sent to the *->browser* and the next step is immediately started. Equally, multiple interactions between the Web *->browser* and WebTransactions are possible within **one and the same** host dialog step.

## array

*->Data type* which can contain a finite set of values of one data type. This data type can be:

- *->scalar*
- a *->class*
- an array

The values in the array are addressed via a numerical index, starting at 0.

## asynchronous message

In WebTransactions, an asynchronous message is one sent to the terminal without having been explicitly requested by the user, i.e. without the user having pressed a key or clicked on an interface element.

## attribute

Attributes define the properties of *->objects*.

An attribute can be, for example, the color, size or position of an object or it can itself be an object. Attributes are also interpreted as *->variables* and their values can be queried or modified.

### **Automask template**

A WebTransactions *->template* created by WebLab either implicitly when generating a base directory or explicitly with the command **Generate Automask**. It is used whenever no format-specific template can be identified. An Automask template contains the statements required for dynamically mapping formats and for communication. Different variants of the Automask template can be generated and selected using the system object attribute `AUTOMASK`.

### **base directory**

The base directory is located on the WebTransactions server and forms the basis for a *->WebTransactions application*. The base directory contains the *->templates* and all the files and program references (links) which are necessary in order to run a WebTransactions application.

### **BCAM application name**

Corresponds to the openUTM generation parameter `BCAMAPPL` and is the name of the *->openUTM application* through which *->UPIC* establishes the connection.

### **browser**

Program which is required to call and display *->HTML* pages. Browsers are, for example, Microsoft Internet Explorer or Mozilla Firefox.

### **browser display print**

The WebTransactions browser display print prints the information displayed in the *->browser*.

### **browser platform**

Operating system of the host on which a *->browser* runs as a client for WebTransactions.

### **buffer**

Definition of a record, which is transmitted from a *->service*. The buffer is used for transmitting and receiving messages. In addition there is a specific buffer for storing the *->recognition criteria* and for data for the representation on the screen.

### **capturing**

To enable WebTransactions to identify the received *->formats* at runtime, you can open a *->session* in *->WebLab* and select a specific area for each format and name the format. The format name and *->recognition criteria* are stored in the *->capture database*. A *->template* of the same name is generated for the format. Capturing forms the basis for the processing of format-specific templates for the WebTransactions for OSD and MVS product variants.

**capture database**

The WebTransactions capture database contains all the format names and the associated *->recognition criteria* generated using the *->capturing* technique. You can use *->WebLab* to edit the sequence and recognition criteria of the formats.

**CGI**

(Common Gateway Interface)

Standardized interface for program calls on *->Web servers*. In contrast to the static output of a previously defined *->HTML* page, this interface permits the dynamic construction of HTML pages.

**class**

Contains definitions of the *->properties* and *->methods* of an *->object*. It provides the model for instantiating objects and defines their interfaces.

**class template**

In WebTransactions, a class template contains valid, recurring statements for the entire object class (e.g. input or output fields). Class templates are processed when the *->evaluation operator* or the `toString` method is applied to a *->host data object*.

**client**

Requestors and users of services in a network.

**cluster**

Set of identical *->WebTransactions applications* on different servers which are interconnected to form a load-sharing network.

**communication object**

This controls the connection to an *->host application* and contains information about the current status of the connection, the last data to be received etc.

**conversion tools**

Utilities supplied with WebTransactions. These tools are used to analyze the data structures of *->openUTM applications* and store the information in files. These files can then be used in WebLab as *->format description sources* in order to generate WTML templates and *->FLD files*. COBOL data structures or IFG format libraries form the basis for the conversion tools. The conversion tool for DRIVE programs is supplied with the product DRIVE.

**daemon**

Name of a process type in Unix system/POSIX systems which runs in the background and performs no I/O operations at terminals.

### **data access control**

Monitoring of the accesses to data and ->*objects* of an application.

### **data type**

Definition of the way in which the contents of a storage location are to be interpreted. Each data type has a name, a set of permitted values (value range), and a defined number of operations which interpret and manipulate the values of that data type.

### **dialog**

Describes the entire communication between browser, WebTransactions and ->*host application*. It will usually comprise multiple ->*dialog cycles*. WebTransactions supports a number of different types of dialog.

- ->*passive dialog*
- ->*active dialog*
- ->*synchronized dialog*
- ->*non-synchronized dialog*

### **dialog cycle**

Cycle that comprises the following steps when a ->*WebTransactions application* is executed:

- construct an ->*HTML* page and send it to the ->*browser*
- wait for a response from the browser
- evaluate the response fields and possibly send them to the ->*host application* for further processing

A number of dialog cycles are passed through while a ->*WebTransactions application* is executing.

### **distinguished name**

The Distinguished Name (DN) in ->*LDAP* is hierarchically organized and consists of a number of different components (e.g. "country, and below country: organization, and below organization: organizational unit, followed by: usual name"). Together, these components provide a unique identification of an object in the directory tree.

Thanks to this hierarchy, the unique identification of objects is a simple matter even in a worldwide directory tree:

- The DN "Country=DE/Name=Emil Person" reduces the problem of achieving a unique identification to the country DE (=Germany).
- The DN "Organization=FTS/Name=Emil Person" reduces it to the organization FTS.
- The DN "Country=DE/Organization=FTS/Name=Emil Person" reduces it to the organization FTS located in Germany (DE).



**document directory**

->*Web server* directory containing the documents that can be accessed via the network. WebTransactions stores files for download in this directory, e.g. the WebLab client or general start pages.

**Domain Name Service (DNS)**

Procedure for the symbolic addressing of computers in networks. Certain computers in the network, the DNS or name server, maintain a database containing all the known host names and *IP numbers* in their environment.

**dynamic data**

In WebTransactions, dynamic data is mapped using the WebTransactions object model, e.g. as a ->*system object*, host object or user input at the browser.

**EHLLAPI****Enhanced High-Level Language API**

Program interface, e.g. of terminal emulations for communication with the SNA world. Communication between the transit client and SNA computer, which is handled via the TRANSIT product, is based on this interface.

**EJB****(Enterprise JavaBean)**

This is a Java-based industry standard which makes it possible to use in-house or commercially available server components for the creation of distributed program systems within a distributed, object-oriented environment.

**entry page**

The entry page is an ->*HTML page* which is required in order to start a ->*WebTransactions application*. This page contains the call which starts WebTransactions with the first ->*template*, the so-called start template.

**evaluation operator**

In WebTransactions the evaluation operator replaces the addressed ->*expressions* with their result (object attribute evaluation). The evaluation operator is specified in the form ##*expression*#.

**expression**

A combination of ->*literals*, ->*variables*, operators and expressions which return a specific result when evaluated.

**FHS****Format Handling System**

Formatting system for BS2000/OSD applications.

**field**

A field is the smallest component of a service and element of a *->record* or *->buffer*.

**field file (\*.fld file)**

In WebTransactions, this contains the structure of a *->format* record (metadata).

**filter**

Program or program unit (e.g. a library) for converting a given *->format* into another format (e.g. XML documents to *->WTScript* data structures).

**format**

Optical presentation on alphanumeric screens (sometimes also referred to as screen form or mask).

In WebTransactions each format is represented by a *->field file* and a *->template*.

**format type**

(only relevant in the case of *->FHS* applications and communication via *->UPIC*) Specifies the type of format: #format, +format, -format or \*format.

**format description sources**

Description of multiple *->formats* in one or more files which were generated from a format library (FHS/IFG) or are available directly at the *->host* for the use of “expressive” names in formats.

**function**

A function is a user-defined code unit with a name and *->parameters*. Functions can be called in *->methods* by means of a description of the function interface (or signature).

**holder task**

A process, a task or a thread in WebTransactions depending on the operating system platform being used. The number of tasks corresponds to the number of users. The task is terminated when the user logs off or when a time-out occurs. A holder task is identical to a *->WebTransactions session*.

**host**

The computer on which the *->host application* is running.

**host adapter**

Host adapters are used to connect existing *->host applications* to WebTransactions. At runtime, for example, they have the task of establishing and terminating connections and converting all the exchanged data.

**host application**

Application that is integrated with WebTransactions.

**host control object**

In WebTransactions, host control objects contain information which relates not to individual fields but to the entire *->format*. This includes, for example, the field in which the cursor is located, the current function key or global format attributes.

**host data object**

In WebTransactions, this refers to an *->object* of the data interface to the *->host application*. It represents a field with all its field attributes. It is created by WebTransactions after the reception of host application data and exists until the next data is received or until termination of the *->session*.

**host data print**

During WebTransactions host data print, information is printed that was edited and sent by the *->host application*, e.g. printout of host files.

**host platform**

Operating system of the host on which the *->host applications* runs.

**HTML**

(Hypertext Markup Language)

See *->Hypertext Markup Language*

**HTTP**

(Hypertext Transfer Protocol)

This is the protocol used to transfer *->HTML* pages and data.

**HTTPS**

(Hypertext Transfer Protocol Secure)

This is the protocol used for the secure transfer of *->HTML* pages and data.

**hypertext**

Document with links to other locations in the same or another document. Users click the links to jump to these new locations.

**Hypertext Markup Language**

(Hypertext Markup Language)

Standardized markup language for documents on the Web.

### Java Bean

Java programs (or *->classes*) with precisely defined conventions for interfaces that allow them to be reused in different applications.

### KDCDEF

openUTM tool for generating *->openUTM applications*.

### LDAP

(Lightweight **D**irectory **A**ccess **P**rotocol)

The X.500 standard defines DAP (Directory Access Protocol) as the access protocol. However, the Internet standard “LDAP” has proved successful specifically for accessing X.500 directory services from a PC.

LDAP is a simplified DAP protocol that does not support all the options available with DAP and is not compatible with DAP. Practically all X.500 directory services support both DAP and LDAP. In practice, interpretation problems may arise since there are various dialects of LDAP. The differences between the dialects are generally small.

### literal

Character sequence that represents a fixed value. Literals are used in source programs to specify constant values (“literal” values).

### master template

WebTransactions template used to generate the Automask and the format-specific templates.

### message queuing (MQ)

A form of communication in which messages are not exchanged directly, rather via intermediate queues. The sender and receiver can work at separate times and locations. Message transmission is guaranteed regardless of whether or not a network connection currently exists.

### method

Object-oriented term for a *->function*. A method is applied to the *->object* in which it is defined.

### module template

In WebTransactions, a module template is used to define *->classes*, *->functions* and constants globally for a complete *->session*. A module template is loaded using the `import()` function.

### MT tag

(Master Template tag)

Special tags used in the dynamic sections of *->master templates*.

**multitier architecture**

All client/server architectures are based on a subdivision into individual software components which are also known as layers or tiers. We speak of 1-tier, 2-tier, 3-tier and multitier models. This subdivision can be considered at the physical or logical level:

- We speak of logical software tiers when the software is subdivided into modular components with clear interfaces.
- Physical tiers occur when the (logical) software components are distributed across different computers in the network.

With WebTransactions, multitier models are possible both at the physical and logical level.

**name/value pair**

In the data sent by the *->browser*, the combination, for example, of an *->HTML* input field name and its value.

**non-synchronized dialog**

Non-synchronized dialogs in WebTransactions permit the temporary deactivation of the checking mechanism implemented in *->synchronized dialogs*. In this way, *->dialogs* that do not form part of the synchronized dialog and have no effect on the logical state of the *->host application* can be incorporated. In this way, for example, you can display a button in an *->HTML* page that allows users to call help information from the current host application and display it in a separate window.

**object**

Elementary unit in an object-oriented software system. Every object possesses a name via which it can be addressed, *->attributes*, which define its status together with the *->methods* that can be applied to the object.

**openUTM**

(Universal Transaction Monitor)

Transaction monitor from Fujitsu Technology Solutions, which is available for BS2000/OSD and a variety of Unix platforms and Windows platforms.

**openUTM application**

A *->host application* which provides services that process jobs submitted by *->clients* or other *->host applications*. openUTM responsibilities include transaction management and the management of communication and system resources. Technically speaking, the UTM application is a group of processes which form a logical unit at runtime.

openUTM applications can communicate both via the client/server protocol *->UPIC* and via the emulation interface (9750).

### **openUTM-Client (UPIC)**

The openUTM-Client (UPIC) is a product used to create client programs for openUTM. openUTM-Client (UPIC) is available, for example, for Unix platforms, BS2000/OSD platforms and Windows platforms.

### **openUTM program unit**

The services of an *->openUTM application* are implemented by one or more openUTM program units. These can be addressed using transaction codes and contain special openUTM function calls (e.g. KDCS calls).

### **parameter**

Data which is passed to a *->function* or a *->method* for processing (input parameter) or data which is returned as a result of a function or method (output parameter).

### **passive dialog**

In the case of passive dialogs in WebTransactions, the dialog sequence is controlled by the *->host application*, i.e. the host application determines the next *->template* which is to be processed. Users who access the host application via WebTransactions pass through the same dialog steps as if they were accessing it from a terminal. WebTransactions uses passive dialog control for the automatic conversion of the host application or when each host application format corresponds to precisely one individual template.

### **password**

String entered for a *->user id* in an application which is used for user authentication (*->system access control*).

### **polling**

Cyclical querying of state changes.

### **pool**

In WebTransactions, this term refers to a shared directory in which WebLab can create and maintain *->base directories*. You control access to this directory with the administration program.

### **post**

To send data.

### **posted object (wt\_Posted)**

List of the data returned by the *->browser*. This *->object* is created by WebTransactions and exists for the duration of a *->dialog cycle*.

**process**

The term “process” is used as a generic term for process (in Solaris, Linux and Windows) and task (in BS2000/OSD).

**project**

In the WebTransactions development environment, a project contains various settings for a ->*WebTransactions application*. These are saved in a project file (suffix *.wtp*). You should create a project for each WebTransactions application you develop, and always open this project for editing.

**property**

Properties define the nature of an ->*object*, e.g. the object “Customer” could have a customer name and number as its properties. These properties can be set, queried, and modified within the program.

**protocol**

Agreements on the procedural rules and formats governing communications between remote partners of the same logical level.

**protocol file**

- openUTM-Client: File into which the openUTM error messages as are written in the case of abnormal termination of a conversation.
- In WebTransactions, protocol files are called trace files.

**roaming session**

->*WebTransactions sessions* which are invoked simultaneously or one after another by different ->*clients*.

**record**

A record is the definition of a set of related data which is transferred to a ->*buffer*. It describes a part of the buffer which may occur one or more times.

**recognition criteria**

Recognition criteria are used to identify ->*formats* of a ->*terminal application* and can access the data of the format. The recognition criteria selected should be one or more areas of the format which uniquely identify the content of the format.

**scalar**

->*variable* made up of a single value, unlike a ->*class*, an ->*array* or another complex data structure.

### service (openUTM)

In *->openUTM*, this is the processing of a request using an *->openUTM application*. There are dialog services and asynchronous services. The services are assigned their own storage areas by openUTM. A service is made up of one or more *->transactions*.

### service application

*->WebTransactions session* which can be called by various different users in turn.

### service node

Instance of a *->service*. During development and runtime of a *->method* a service can be instantiated several times. During modelling and code editing those instances are named service nodes.

### session

When an end user starts to work with a *->WebTransactions application* this opens a WebTransactions session for that user on the WebTransactions server. This session contains all the connections open for this user to the *->browsers*, special *->clients* and *->hosts*.

A session can be started as follows:

- Input of a WebTransactions URL in the browser.
- Using the `START_SESSION` method of the `WT_REMOTE` client/server interface.

A session is terminated as follows:

- The user makes the corresponding input in the output area of this *->WebTransactions application* (not via the standard browser buttons).
- Whenever the configured time that WebTransactions waits for a response from the *->host application* or from the *->browser* is exceeded.
- Termination from WebTransactions administration.
- Using the `EXIT_SESSION` method of the `WT_REMOTE` client/server interface.

A WebTransactions session is unique and is defined by a *->WebTransactions application* and a session ID. During the life cycle of a session there is one *->holder task* for each WebTransactions session on the WebTransactions server.

### SOAP

(originally **S**imple **O**bject **A**ccess **P**rotocol)

The *->XML* based SOAP protocol provides a simple, transparent mechanism for exchanging structured and typecast information between computers in a decentralized, distributed environment.

SOAP provides a modular package model together with mechanisms for data encryption within modules. This enables the uncomplicated description of the internal interfaces of a *->Web-Service*.



**style**

In WebTransactions this produces a different layout for a *->template*, e.g. with more or less graphic elements for different *->browsers*. The style can be changed at any time during a *->session*.

**synchronized dialog**

In the case of synchronized dialogs (normal case), WebTransactions automatically checks whether the data received from the web browser is genuinely a response to the last *->HTML* page to be sent to the *->browser*. For example, if the user at the web browser uses the **Back** button or the History function to return to an “earlier” HTML page of the current *->session* and then returns this, WebTransactions recognizes that the data does not correspond to the current *->dialog cycle* and reacts with an error message. The last page to have been sent to the browser is then automatically sent to it again.

**system access control**

Check to establish whether a user under a particular *->user ID* is authorized to work with the application.

**system object (wt\_System)**

The WebTransactions system object contains *->variables* which continue to exist for the duration of an entire *->session* and are not cleared until the end of the session or until they are explicitly deleted. The system object is always visible and is identical for all name spaces.

**TAC**

See *->transaction code*

**tag**

*->HTML*, *->XML* and *->WTML* documents are all made up of tags and actual content. The tags are used to mark up the documents e.g. with header formats, text highlighting formats (bold, italics) or to give source information for graphics files.

**TCP/IP**

(Transport **C**ontrol **P**rotocol/**I**nternet **P**rotocol)

Collective name for a protocol family in computer networks used, for example, in the Internet.

### template

A template is used to generate specific code. A template contains fixed information parts which are adopted unchanged during generation, as well as variable information parts that can be replaced by the appropriate values during generation.

A template is a *->WTML* file with special tags for controlling the dynamic generation of a *->HTML* page and for the processing of the values entered at the *->browser*. It is possible to maintain multiple template sets in parallel. These then represent different *->styles* (e.g. many/few graphics, use of Java, etc.).

WebTransactions uses different types of template:

- *->Automask templates* for the automatic conversion of the *->formats* of MVS and OSD applications.
- Custom templates, written by the programmer, for example, to control an *->active dialog*.
- Format-specific templates which are generated for subsequent post-processing.
- Include templates which are inserted in other templates.
- *->Class templates*
- *->Master templates* to ensure the uniform layout of fixed areas on the generation of the Automask and format-specific templates.
- Start template, this is the first template to be processed in a WebTransactions application.

### template object

*->Variables* used to buffer values for a *->dialog cycle* in WebTransactions.

### terminal application

Application on a *->host* computer which is accessed via a 9750 or 3270 interface.

### terminal hardcopy print

A terminal hardcopy print in WebTransactions prints the alphanumeric representation of the *->format* as displayed by a terminal or a terminal emulation.

### transaction

Processing step between two synchronization points (in the current operation) which is characterized by the ACID conditions (**A**tomicity, **C**onsistency, **I**solation and **D**urability). The intentional changes to user information made within a transaction are accepted either in their entirety or not at all (all-or-nothing rule).

**transaction code/TAC**

Name under which an openUTM service or ->*openUTM program unit* can be called. The transaction code is assigned to the openUTM program unit during configuration. A program unit can be assigned several transaction codes.

**UDDI**

(**U**niversal **D**escription, **D**iscovery and **I**ntegration)

Refers to directories containing descriptions of ->*Web services*. This information is available to web users in general.

**Unicode**

An alphanumeric character set standardized by the International Standardisation Organisation (ISO) and the Unicode Consortium. It is used to represent various different types of characters: letters, numerals, punctuation marks, syllabic characters, special characters and ideograms. Unicode brings together all the known text symbols in use across the world into a single character set. Unicode is vendor-independent and system-independent. It uses either two-byte or four-byte character sets in which each text symbol is encoded. In the ISO standard, these character sets are termed UCS-2 (Universal Character Set 2) or UCS-4. The designation UTF-16 (Unicode Transformation Format 16-bit), which is a standard defined by the Unicode Consortium, is often used in place of the designation UCS-2 as defined in ISO. Alongside UTF-16, UTF-8 (Unicode Transformation Format 8 Bit) is also in widespread use. UTF-8 has become the character encoding method used globally on the Internet.

**UPIC**

(**U**niversal **P**rogramming **I**nterface for **C**ommunication)

Carrier system for openUTM clients which uses the X/Open interface, which permits CPI-C client/server communication between a CPI-C-Client application and the openUTM application.

**URI**

(**U**niform **R**esource **I**dentifier)

Blanket term for all the names and addresses that reference objects on the Internet. The generally used URIs are->*URLs*.

**URL**

(**U**niform **R**esource **L**ocator)

Description of the location and access type of a resource in the ->*Internet*.

**user exit**

Functions implemented in C/C++ which the programmer calls from a ->*template*.

**user ID**

User identification which can be assigned a password (->*system access control*) and special access rights (->*data access control*).

**variable**

Memory location for variable values which requires a name and a ->*data type*.

**visibility of variables**

->*Objects* and ->*variables* of different dialog types are managed by WebTransactions in different address spaces. This means that variables belonging to a ->*synchronized dialog* are not visible and therefore not accessible in a ->*asynchronous dialog* or in a dialog with a remote application.

**web server**

Computer and software for the provision of ->*HTML* pages and dynamic data via ->*HTTP*.

**web service**

Service provided on the Internet, for example a currency conversion program. The SOAP protocol can be used to access such a service. The interface of a web service is described in ->*WSDL*.

**WebTransactions application**

This is an application that is integrated with ->*host applications* for internet/intranet access. A WebTransactions application consists of:

- a ->*base directory*
- a start template
- the ->*templates* that control conversion between the ->*host* and the ->*browser*.
- protocol-specific configuration files.

**WebTransactions platform**

Operating system of the host on which WebTransactions runs.

**WebTransactions server**

Computer on which WebTransactions runs.

**WebTransactions session**

See ->*session*

**WSDL**

(**Web Service Definition Language**)

Provides ->*XML* language rules for the description of ->*web services*. In this case, the web service is defined by means of the port selection.

## WTBean

In WebTransactions ->*WTML* components with a self-descriptive interface are referred to as WTBeans. A distinction is made between inline and standalone WTBeans:

- An inline WTBean corresponds to a part of a WTML document
- A standalone WTBean is an autonomous WTML document

A number of WTBeans are included in of the WebTransactions product, additional WTBeans can be downloaded from the WebTransactions homepage [ts.fujitsu.com/products/software/openseas/webtransactions.html](http://ts.fujitsu.com/products/software/openseas/webtransactions.html).

## WTML

(WebTransactions Markup Language)

Markup and programming language for WebTransactions ->*templates*. WTML uses additional ->*WTML tags* to extend ->*HTML* and the server programming language ->*WScript*, e.g. for data exchange with ->*host applications*. WTML tags are executed by WebTransactions and not by the ->*browser* (serverside scripting).

## WTML tag

(WebTransactions Markup Language-Tag)

Special WebTransactions tags for the generation of the dynamic sections of an ->*HTML* page using data from the ->*host application*.

## WScript

Serverside programming language of WebTransactions. WScripts are similar to client-side Java scripts in that they are contained in sections that are introduced and terminated with special tags. Instead of using ->*HTML-SCRIPT* tags you use ->*WTML-Tags*: `wtOnCreateScript` and `wtOnReceiveScript`. This indicates that these scripts are to be implemented by WebTransactions and not by the ->*browser* and also indicates the time of execution. OnCreate scripts are executed before the page is sent to the browser. OnReceive scripts are executed when the response has been received from the browser.

## XML

(eXtensible Markup Language)

Defines a language for the logical structuring of documents with the aim of making these easy to exchange between various applications.

## XML schema

An XML schema basically defines the permissible elements and attributes of an XML description. XML schemas can have a range of different formats, e.g. DTD (Document Type Definition), XML Schema (W3C standard) or XDR (XML Data Reduced).



---

## Abbreviations

BO	<b>B</b> usiness <b>O</b> bject
CGI	<b>C</b> ommon <b>G</b> ateway <b>I</b> nterface
DN	<b>D</b> istinguished <b>N</b> ame
DNS	<b>D</b> omain <b>N</b> ame <b>S</b> ervice
EJB	<b>E</b> nterprise <b>J</b> ava <b>B</b> ean
FHS	<b>F</b> ormat <b>H</b> andling <b>S</b> ystem
HTML	<b>H</b> ypertext <b>M</b> arkup <b>L</b> anguage
HTTP	<b>H</b> ypertext <b>T</b> ransfer <b>P</b> rotocol
HTTPS	<b>H</b> ypertext <b>T</b> ransfer <b>P</b> rotocol <b>S</b> ecure
IFG	<b>I</b> nteraktiver <b>F</b> ormat <b>G</b> enerator
ISAPI	<b>I</b> nternet <b>S</b> erver <b>A</b> pplication <b>P</b> rogramming <b>I</b> nterface
LDAP	<b>L</b> ightweight <b>D</b> irectory <b>A</b> ccess <b>P</b> rotocol
LPD	<b>L</b> ine <b>P</b> rinter <b>D</b> aemon
MT-Tag	<b>M</b> aster- <b>T</b> emplate- <b>T</b> ag
MVS	<b>M</b> ultiple <b>V</b> irtual <b>S</b> torage
OSD	<b>O</b> pen <b>S</b> ystems <b>D</b> irection
SGML	<b>S</b> tandard <b>G</b> eneralized <b>M</b> arkup <b>L</b> anguage
SOAP	<b>S</b> imple <b>O</b> bject <b>A</b> ccess <b>P</b> rotocol

## Abbreviations

---

SSL	<b>S</b> ecure <b>S</b> ocket <b>L</b> ayer
TCP/IP	<b>T</b> ransport <b>C</b> ontrol <b>P</b> rotocol/ <b>I</b> nternet <b>P</b> rotocol
Upic	<b>U</b> niversal <b>P</b> rogramming <b>I</b> nterface for <b>C</b> ommunication
URL	<b>U</b> niform <b>R</b> esource <b>L</b> ocator
WSDL	<b>W</b> eb <b>S</b> ervices <b>D</b> escription <b>L</b> anguage
wtc	<b>W</b> eb <b>T</b> ransactions <b>C</b> omponent
WTML	<b>W</b> eb <b>T</b> ransactions <b>M</b> arkup <b>L</b> anguage
XML	<b>e</b> Xtensible <b>M</b> arkup <b>L</b> anguage



---

# Related publications

## WebTransactions manuals

You can download all manuals from the Web address <http://manuals.ts.fujitsu.com>.

**WebTransactions**  
**Concepts and Functions**

Introduction

**WebTransactions**  
**Client APIs for WebTransactions**

User Guide

**WebTransactions**  
**Connection to openUTM Applications via UPIC**

User Guide

**WebTransactions**  
**Connection to OSD Applications**

User Guide

**WebTransactions**  
**Connection to MVS Applications**

User Guide

**WebTransactions**  
**Access to Dynamic Web Contents**

User Guide

**WebTransactions**  
**Web Frontend for Web Services**

User Guide

## Other publications

The manuals are available as online manuals, see <http://manuals.ts.fujitsu.com>, or in printed form which must be paid and ordered separately at <http://manualshop.ts.fujitsu.com>.

### ***interNet Services***

Administrator Guide

---

# Index

.clt 309

## A

abs (Math class) 161

access

class elements 343

acos (Math class) 162

activate

Java support 338

active dialog 381, 384

add (WT\_LdapConnection class) 237

addition 65

ArchiveName tag 332

arithmetic operators 65

array 381

Java object 347

multidimensional 347

Array class 118

concat 121

equals 122

getClassName 123

join 124

pop 125

push 126

reverse 127

shift 128

slice 129

sort 130

splice 133

toString 134

unshift 135

valueOf 136

asin (Math class) 162

assignment 278

assignment operators 71

asynchronous message 381

atan (Math class) 163

attribute 381

read and modify 346

automask template 382

## B

base data type 381

base directory 382

BCAM application name 382

BCAMAPPL 382

BinaryFile tag 332

bind (WT\_LdapConnection class) 238

bindSasl (WT\_LdapConnection class) 239

bitwise operators 68

block 281, 283, 286

Boolean class 137

equals 137

getClassName 138

setValue 138

toString 139

valueOf 140

boolean data type 45

boolean operators 70

branch 279

branch destination see label

break 290

do/while loop 283

for loop 284

for/in loop 286

while loop 281

browser 382

browser display print 382

browser platform 382

buffer [382](#)

### C

C/C++ user exits [350](#)

    defining [351](#)

    examples [353](#)

    link [351](#)

    supplied files [350](#)

capture database [383](#)

capturing [382](#)

case [288](#)

catch block [305](#)

ceil (Math class) [163](#)

CGI (Common Gateway Interface) [383](#)

change

    styles (example) [375](#)

character set [29](#)

charAt (String class) [192](#)

charCodeAt (String class) [193](#)

checkLogin [357](#)

checkProcess [357](#)

class [117](#), [383](#)

    Array [118](#)

    Boolean [137](#)

    Date [141](#)

    Document [148](#)

    Function [157](#)

    host data object [154](#)

    Math [161](#)

    Number [170](#)

    Object [173](#)

    RegExp [180](#)

    String [191](#)

    templates [383](#)

    WT\_Communication [211](#)

    WT\_Filter [216](#)

    WT\_LdapConnection [234](#)

    WT\_Userexit [258](#)

class element

    access [343](#)

Class tag [331](#)

class template [309](#)

clear (Document class) [149](#)

client [383](#)

close (Document class) [149](#)

close (WT\_Communication class) [212](#)

cluster [383](#)

code

    guarded section [305](#)

comma operator [74](#)

comment

    JavaScript format [31](#)

comment tag [264](#)

CommObj tag [330](#)

communication object [211](#), [383](#)

compare (WT\_LdapConnection class) [240](#)

comparison operators [66](#)

compile (RegExp class) [184](#)

concat (Array class) [121](#)

concat (String class) [194](#)

condition [279](#), [281](#), [283](#), [284](#)

condition operator [73](#)

conditional branch [277](#)

conditional execution [279](#)

configuration

    JVM see define parameters for the Java Virtual  
    Machine

continue [292](#)

    do/while loop [283](#)

    for loop [284](#)

    for/in loop [286](#)

    while loop [281](#)

control structures [279](#)

conversion tools [383](#)

copyFile() [83](#)

cos (Math class) [164](#)

create

    Java object in WTSript [341](#)

createFolder() [84](#)

Creationtime [358](#)

### D

daemon [383](#)

data

    dynamic [385](#)

data access control [384](#)

data type [44](#), [384](#)

    boolean [45](#), [281](#)

- conversion [295](#)
- function [46](#)
- number [45](#)
- object [46](#)
- string [46](#)
- stringlike [47](#)
- undefined [45](#)
- database
  - information (LDAP) [234](#)
- Dataform tag [265](#)
- dataObjectToFormattedXML (WT\_Filter class) [219](#)
- dataObjectToXML (WT\_Filter class) [217](#)
- Date class [141](#)
  - equals [142](#)
  - getClassName [142](#)
  - getDay [143](#)
  - getHours [143](#)
  - getMinutes [143](#)
  - getMonth [143](#)
  - getSeconds [143](#)
  - getTimezoneOffset [144](#)
  - getYear [143](#)
  - setDay [145](#)
  - setHours [145](#)
  - setMinutes [145](#)
  - setMonth [145](#)
  - setSeconds [145](#)
  - setYear [145](#)
  - toGMTString [145](#)
  - toLocaleString [146](#)
  - toString [146](#)
  - valueOf [147](#)
- declare
  - function [297](#)
  - variable [295](#)
- decrement [65](#)
- decrement operator [278](#)
- default object for statement [301](#)
- define
  - parameters for the Java Virtual Machine [340](#)
- define/initialize counter [285](#)
- delete
  - operator [76](#)
  - deleteEntry (WT\_LdapConnection class) [241](#)
  - deleteFile() [85](#)
  - Delfile [359](#)
  - dialog [384](#)
    - active [384](#)
    - non-synchronized [384, 389](#)
    - passive [384, 390](#)
    - synchronized [384, 393](#)
    - types [384](#)
  - dialog cycle [384](#)
  - directory
    - LDAP [234](#)
    - lookup operation (LDAP) [234](#)
    - search operation (LDAP) [234](#)
  - directory service
    - provide (LDAP) [234](#)
  - directory service protocol
    - LDAP [235](#)
  - directory tree
    - LDAP [235](#)
  - distinguished name [235, 384](#)
  - division [65](#)
  - DN (LDAP)
    - distinguished name [235](#)
  - do [283](#)
  - DO UNTIL tag [272](#)
  - DO WHILE tag [271](#)
  - do/while loop [283](#)
  - Document class [148](#)
    - clear [149](#)
    - close [149](#)
    - equals [150](#)
    - getClassName [150](#)
    - open [151](#)
    - read [152](#)
    - valueOf [152](#)
    - write [153](#)
    - writeln [153](#)
  - document directory [385](#)
  - document.write method [285](#)
  - Domain Name Service (DNS) [385](#)
  - dynamic pages
    - without host application [335](#)

### E

EHLLAPI [385](#)  
EJB [385](#)  
else [279](#)  
entries  
    LDAP [235](#)  
entry page [385](#)  
environment variables  
    Java [340](#)  
equals (Array class) [122](#)  
equals (Boolean class) [137](#)  
equals (Date class) [142](#)  
equals (Document class) [150](#)  
equals (Function class) [160](#)  
equals (Number class) [171](#)  
equals (Object class) [174](#)  
equals (RegExp class) [185](#)  
equals (String class) [195](#)  
equals (WT\_Communication class) [212](#)  
equals (WT\_LdapConnection class) [241](#)  
error object [302](#)  
Escape sequence (in strings) [35](#)  
escape() [86](#)  
euro symbol [318](#)  
eval() [87](#)  
evaluate() [88](#)  
evaluation operator [26](#), [385](#)  
exception [302](#)  
    explicit [304](#)  
    handling [302](#), [305](#)  
exec (RegExp class) [186](#)  
Exit tag [267](#)  
exitDialogStep() [90](#)  
exitReceiveProcessing() [91](#)  
exitScript() [92](#)  
exitSession() [94](#)  
exitTemplate() [95](#)  
exp (Math class) [164](#)  
explicit exception [304](#)  
explicit variable declaration [295](#)  
explodeDn (WT\_LdapConnection class) [242](#)  
expression [288](#), [385](#)  
expressions [63](#)  
    as statements [278](#)

### F

FHS [385](#)  
field [386](#)  
field file [386](#)  
filter [386](#)  
finally block [305](#)  
firstEntry (WT\_LdapConnection class) [243](#)  
fld file [386](#)  
floating-point values [34](#)  
floor (Math class) [165](#)  
for [284](#)  
for/in [286](#)  
for/in loop [285](#)  
format [386](#)  
    #format [386](#)  
    \*format [386](#)  
    +format [386](#)  
    -format [386](#)  
format description source [386](#)  
Format tag [330](#)  
format type [386](#)  
forward() [96](#)  
FreeBuffer [359](#)  
FreeNameInPool [359](#)  
fromCharCode (String class) [195](#)  
fully qualified specification [54](#)  
function [297](#), [300](#), [301](#), [386](#)  
    copyFile() [83](#)  
    createFolder() [84](#)  
    deleteFile() [85](#)  
    escape() [86](#)  
    eval() [87](#)  
    evaluate() [88](#)  
    exitDialogStep() [90](#)  
    exitReceiveProcessing() [91](#)  
    exitScript() [92](#)  
    exitSession() [94](#)  
    exitTemplate() [95](#)  
    forward() [96](#)  
    global [83](#)  
    import() [98](#)  
    include() [99](#)  
    isRequestWaiting() [102](#)  
    listFolder() [104](#)

- local variable [286](#)
- moveFile() [106](#)
- Number() [107](#)
- parseFloat() [108](#)
- parseInt() [109](#)
- setNextPage() [110](#)
- setSingleStep() [111](#)
- setTimeout() [112](#)
- setTraceLevel() [113](#)
- String() [113](#)
- unescape() [114](#)
- writeToTrace() [115](#)
- Function class [157](#)
  - equals [160](#)
  - getClassName [160](#)
- function data type [46](#)
- function declaration [277](#)
- functions
  - deleteFile() [85](#)
  - setSingleStep() [111](#)
- G**
- GenerationInfo tag [329](#)
- getClassName (Array class) [123](#)
- getClassName (Boolean class) [138](#)
- getClassName (Date class) [142](#)
- getClassName (Document class) [150](#)
- getClassName (Function class) [160](#)
- getClassName (host data object class) [154](#)
- getClassName (Number class) [171](#)
- getClassName (Object class) [174](#)
- getClassName (RegExp class) [189](#)
- getClassName (String class) [196](#)
- getClassName (WT\_Communication class) [213](#)
- getClassName (WT\_LdapConnection class) [244](#)
- Getdate [360](#)
- getDay (Date class) [143](#)
- Getdir [360](#)
- getDn (WT\_LdapConnection class) [244](#)
- getEntries (WT\_LdapConnection class) [245](#)
- Getfile [360](#)
- getHours (Date class) [143](#)
- GetInstallDir [361](#)
- getMinutes (Date class) [143](#)
- getModule (WT\_Communication class) [213](#)
- getMonth (Date class) [143](#)
- getOption (WT\_LdapConnection class) [246](#)
- getSeconds (Date class) [143](#)
- Gettime [361](#)
- getTimezoneOffset (Date class) [144](#)
- getYear (Date class) [143](#)
- global functions [83](#)
- global services (LDAP) [234](#)
- global variable [49](#), [295](#)
- guarded code section [305](#)
- H**
- hierarchical tree structure [235](#)
- holder task [386](#)
- host [386](#)
- host adapter [386](#)
- host application [387](#)
- host control object [387](#)
- host data object [387](#)
- host data object class [154](#)
  - getClassName [154](#)
  - toString [155](#)
  - valueOf [156](#)
- host data print [387](#)
- host platform [387](#)
- HTML [387](#)
  - short reference guide [379](#)
- HTML area [24](#)
- HTML editor [25](#)
- HTML tag, static output [24](#)
- HTTP [387](#)
- HTTPS [387](#)
- hypertext [387](#)
- Hypertext Markup Language (HTML) [387](#)
- I**
- identifier [295](#), [297](#)
- if [279](#)
- IF tag [269](#)
- implicit variable declaration [295](#)
- import() [98](#)
- in operator [76](#)
- include tag [268](#)

- include() [99](#)
  - using in a function [100](#)
- increment [65](#)
- increment operator [278](#)
- index operator [53](#)
- indexOf (String class) [197](#)
- information
  - database (LDAP) [234](#)
- init [284](#)
- initialization [53](#)
- inline WtBean [397](#)
- instanceOf operator [77](#)
- invoke
  - Java method in WtScript [344](#)
- isRequestWaiting() [102](#)
  
- J**
- Java Bean [388](#)
- Java integration
  - environment variables [340](#)
  - example [349](#)
  - exception handling [346](#)
  - read and modify attributes [346](#)
  - system attributes [340](#)
  - using arrays [347](#)
- Java method
  - invoke in WtScript [344](#)
- Java object
  - array [347](#)
  - create in WtScript [341](#)
  - using in WtScript [342](#)
- Java runtime environment [338](#)
- Java support
  - activating [338](#)
- Java user exits [338](#)
  - environment variables [340](#)
  - system attributes [340](#)
- Java Virtual Machine (JVM)
  - define parameters [340](#)
- JAVA\_CHECK\_SOURCE
  - environment variable (Java) [340](#)
- JAVA\_CLASSPATH
  - environment variable (Java) [340](#)
- JAVA\_DEBUG
  - environment variable (Java) [340](#)
- JAVA\_DEBUG\_PORT
  - environment variable (Java) [340](#)
- JAVA\_DISABLE\_ASYNC\_GC
  - environment variable (Java) [340](#)
- JAVA\_DISABLE\_CLASS\_GC
  - environment variable (Java) [340](#)
- JAVA\_ENABLE\_VERBOSE\_GC
  - environment variable (Java) [340](#)
- JAVA\_INITIAL\_HEAP\_SIZE
  - environment variable (Java) [340](#)
- JAVA\_MAX\_HEAP\_SIZE
  - environment variable (Java) [340](#)
- JAVA\_NATIVE\_STACK\_SIZE
  - environment variable (Java) [340](#)
- JAVA\_STACK\_SIZE
  - environment variable (Java) [340](#)
- JAVA\_VERBOSE
  - environment variable (Java) [340](#)
- JAVA\_VERIFY\_MODE
  - environment variable (Java) [340](#)
- JavaScript
  - client-side [25](#)
  - server-side [277](#)
- join (Array class) [124](#)
- JVM see Java Virtual Machine
  
- K**
- KDCDEF [388](#)
- keywords [32](#)
  - this [77](#)
  
- L**
- label [281](#), [283](#), [284](#), [286](#), [288](#), [290](#), [292](#)
- lastIndexOf (String class) [198](#)
- LDAP [234](#), [388](#)
  - directory [234](#)
  - directory service protocol [235](#)
  - directory tree [235](#)
  - entries [235](#)
  - error messages [235](#)
  - functionality [235](#)
  - overview [234](#)



- see WT\_LdapConnection class
  - tree structure [235](#)
- length (Array class) [120](#)
- length (String class) [191](#)
- lexical elements [29](#)
- libWTHolderUTM.a [352](#)
- lifetime
  - predefined object [51](#)
  - variable [51](#)
- line-end characters [31](#)
- Lines tag [315](#)
- listFolder() [104](#)
- literals [33](#), [388](#)
  - floating-point values [34](#)
  - logical values [36](#)
  - natural numbers [34](#)
  - null object [37](#)
  - regular expressions [38](#)
  - string [35](#)
  - text [33](#)
- local variable [49](#), [295](#)
- LockNameInPool [362](#)
- log (Math class) [165](#)
- logical values [36](#)
- lookup operation
  - directory (LDAP) [234](#)
- loop [277](#)
  - end [290](#)
  - repeat execution [292](#)
- loop counter [285](#)
- loose typing [43](#), [295](#)
- M**
- make file (for user exits) [352](#)
- master template [313](#), [388](#), [394](#)
  - standard [313](#)
  - tag [388](#)
- match (String class) [199](#)
- Math class [161](#)
  - abs [161](#)
  - acos [162](#)
  - asin [162](#)
  - atan [163](#)
  - ceil [163](#)
  - cos [164](#)
  - exp [164](#)
  - floor [165](#)
  - log [165](#)
  - max [166](#)
  - min [166](#)
  - pow [167](#)
  - random [167](#)
  - round [168](#)
  - sin [169](#)
  - sqrt [169](#)
  - tan [169](#)
- max (Math class) [166](#)
- MAX\_VALUE [170](#)
- message queuing [388](#)
- method [117](#), [388](#)
  - abs (Math class) [161](#)
  - acos (Math class) [162](#)
  - add (WT\_LdapConnection class) [237](#)
  - asin (Math class) [162](#)
  - atan (Math class) [163](#)
  - bind (WT\_LdapConnection class) [238](#)
  - bindSasl (WT\_LdapConnection class) [239](#)
  - ceil (Math class) [163](#)
  - charAt (String class) [192](#)
  - charCodeAt (string class) [193](#)
  - clear (Document class) [149](#)
  - close (Document class) [149](#)
  - close (WT\_Communication class) [212](#)
  - compare (WT\_LdapConnection class) [240](#)
  - compile (RegExp class) [184](#)
  - concat (Array class) [121](#)
  - concat (String class) [194](#)
  - cos (Math class) [164](#)
  - dataObjectToFormattedXML (WT\_Filter class) [219](#)
  - dataObjectToXML (WT\_Filter class) [217](#)
  - deleteEntry (WT\_LdapConnection class) [241](#)
  - equals (Array class) [122](#)
  - equals (Boolean class) [137](#)
  - equals (Date class) [142](#)
  - equals (Document class) [150](#)
  - equals (Function class) [160](#)

- equals (Number class) 171
- equals (Object class) 174
- equals (RegExp class) 185
- equals (String class) 195
- equals (WT\_Communication class) 212
- equals (WT\_LdapConnection class) 241
- exec (RegExp class) 186
- exp (Math class) 164
- explodeDn (WT\_LdapConnection class) 242
- firstEntry (WT\_LdapConnection class) 243
- floor (Math class) 165
- fromCharCode (string class) 195
- getClassName (Array class) 123
- getClassName (Boolean class) 138
- getClassName (Date class) 142
- getClassName (Document class) 150
- getClassName (Function class) 160
- getClassName (host data object class) 154
- getClassName (Number class) 171
- getClassName (Object class) 174
- getClassName (RegExp class) 189
- getClassName (String class) 196
- getClassName (WT\_Communication class) 213
- getClassName (WT\_LdapConnection class) 244
- getDay (Date class) 143
- getEntries (WT\_LdapConnection class) 245
- getHours (Date class) 143
- getMinutes (Date class) 143
- getModule (WT\_Communication class) 213
- getMonth (Date class) 143
- getOption (WT\_LdapConnection class) 246
- getSeconds (Date class) 143
- getTimezoneOffset (Date class) 144
- getYear (Date class) 143
- indexOf (String class) 197
- join (Array class) 124
- lastIndexOf (String class) 198
- log (Math class) 165
- match (String class) 199
- max (Math class) 166
- methodCallToXML (WT\_Filter class) 222
- min (Math class) 166
- modify (WT\_LdapConnection class) 247
- nextEntry (WT\_LdapConnection class) 248
- objectTreeToXML (WT\_Filter class) 223
- open (Document class) 151
- open (WT\_Communication class) 214
- pop (Array class) 125
- pow (Math class) 167
- push (Array class) 126
- random (Math class) 167
- read (Document class) 152
- receive (WT\_Communication class) 214
- replace (String class) 201
- reverse (Array class) 127
- round (Math class) 168
- search (String class) 203
- search (WT\_LdapConnection class) 249
- send (WT\_Communication class) 215
- setDay (Date class) 145
- setMinutes (Date class) 145
- setMonth (Date class) 145
- setOption (WT\_LdapConnection class) 253
- setSeconds (Date class) 145
- setValue (Boolean class) 138
- setValue (Number class) 172
- setValue (String class) 204
- setYear (Date class) 145
- shift (Array class) 128
- sin (Math class) 169
- slice (Array class) 129
- slice (String class) 205
- sort (Array class) 130
- splice (Array class) 133
- split (String class) 206
- sqrt (Math class) 169
- substr (String class) 207
- substring (String class) 208
- tan (Math class) 169
- test (RegExp class) 190
- toGMTString (Date class) 145
- toLocaleString (Date class) 146
- toLowerCase (String class) 209
- toString (Array class) 134
- toString (Boolean class) 139
- toString (Date class) 146

- toString (host data object class) 155
- toString (Number class) 172
- toString (String class) 209
- toString (WT\_LdapConnection class) 254
- toUpperCase (String class) 210
- unshift (Array class) 135
- user functions (WT\_Userexit class) 258
- valueOf (Array class) 136
- valueOf (Boolean class) 140
- valueOf (Date class) 147
- valueOf (Document class) 152
- valueOf (host data object class) 156
- valueOf (Number class) 172
- valueOf (Object class) 179
- valueOf (String class) 210
- valueOf (WT\_LdapConnection class) 255
- write (Document class) 153
- writeln (Document class) 153
- XML\_SAXParse (WT\_Filter class) 227
- XMLToDataObject (WT\_Filter class) 224
- XMLToMethodCall (WT\_Filter class) 225
- XMLToObjectTree (WT\_Filter class) 226
- methodCallToXML (WT\_Filter class) 222
- MethodInterface tag 333
- methods
  - getDay (Date class) 143
  - getDn (WT\_LdapConnection class) 244
  - random (Math class) 167
  - setHours (Date class) 145
  - toString (Object class) 175
  - unbind (WT\_LdapConnection class) 254
- min (Math class) 166
- MIN\_VALUE 170
- Modificationtime 362
- modify
  - attribute 346
- modify (WT\_LdapConnection class) 247
- module template 388
- modulus 65
- moveFile() 106
- MT tag 314, 388
  - ArchiveName 332
  - BinaryFile 332
  - Class 331
- CommObj 330
- Format 330
- GenerationInfo 329
- Lines 315
- MethodInterface 333
- NationalVariant 330
- ObjectName 331
- OnReceiveCopies 327, 328
- Options 321, 323
- PackageName 332
- Source 331
- multidimensional array 347
- multiplication 65
- multitier architecture 389
- N**
- name elements 42
- name in expression 286
- name/value pair 389
- names
  - assigning objects 56
  - fully qualified specification 54
  - overriding (variable) 49
  - overview of name spaces 52
  - relative specification 55
  - structure 53
- NaN 170
- NationalVariant tag 330
- natural numbers 34
- new operator 75
- nextEntry (WT\_LdapConnection class) 248
- non-synchronized dialog 384, 389
- null object 37
- Number class 170
  - equals 171
  - getClassName 171
  - setValue 172
  - toString 172
  - valueOf 172
- number data type 45
- Number() 107
- O**
- object 217, 301, 389

- Object class 173
  - equals 174
  - getClassName 174
  - toString 175
  - valueOf 179
- object data type 46
- object reference 277, 301
- ObjectName tag 331
- objectTreeToXML (WT\_Filter class) 223
- OnCreateScript tag 274, 277
- OnReceiveCopies tag 327, 328
- OnReceiveScript tag 275, 277
- open (Document class) 151
- open (WT\_Communication class) 214
- openUTM 389
  - application 389
  - Client 390
  - program unit 390
  - service 392
- operations 384
- operators
  - arithmetic 65
  - assignment 71
  - bitwise 68
  - boolean 70
  - comma 74
  - comparison 66
  - condition 73
  - deleting 76
  - in 76
  - instanceOf 77
  - new 75
  - string concatenation 72
  - WT\_THIS 77
- Options tag 321
  - extended syntax 323
  - standard syntax 321
- P**
- PackageName tag 332
- parameter 297, 300, 390
- parameter transfer
  - when invoking Java method in WTScrip 344
- parseFloat() 108
- parseInt() 109
- pass parameter 294
- pass value to function 297
- passive dialog 384, 390
- password 390
- point operator 53
- poll Exit button (example) 376
- polling 390
- pool 390
- pop (Array class) 125
- posted object 390
- posting 390
- pow (Math class) 167
- predefined objects, lifetime 51
- process 391
- project 391
- property 391
- protocol 391
- protocol file 391
- provide
  - directory service (LDAP) 234
- pseudo tag see WTMML tag
- push (Array class) 126
- Putfile 363
- R**
- random (Math class) 167
- read
  - attribute 346
- read (Document class) 152
- receive (WT\_Communication class) 214
- recognition criteria 391
- record 391
- record structure 386
- reference data type 44
- RegExp (predefined object) 183
- RegExp class 180
  - compile 184
  - equals 185
  - exec 186
  - getClassName 189
  - test 190
- regular expressions 38
- relative specification 55

- ReleaseStationName 363
- Rem 264, 379
- Rem tag 264
- replace (String class) 201
- ReplaceByConfigFile 364
- ReserveStationName 364
- return 294
  - result of function 294
- retValue 294
- reverse (Array class) 127
- round (Math class) 168
  
- S**
- scalar 391
- search (String class) 203
- search (WT\_LdapConnection class) 249
- search operation
  - directory (LDAP) 234
- send (WT\_Communication class) 215
- SendMail 365
- separators 31
- sequence control 277, 279
- service
  - global (LDAP) 234
- service (openUTM) 392
- service node 392
- session 392
  - terminating 267
  - terminating (exitSession()) 94
  - WebTransactions 392
- setDay (Date class) 145
- setHours (Date class) 145
- setMinutes (Date class) 145
- setMonth (Date class) 145
- setNextPage() 110
- setOption (WT\_LdapConnection class) 253
- setSeconds (Date class) 145
- setSingleStep() 111
- setTimeout() 112
- setTraceLevel() 113
- setValue (Boolean class) 138
- setValue (Number class) 172
- setValue (String class) 204
- setYear (Date class) 145
  
- shift (Array class) 128
- short reference guide
  - HTML 379
  - WTML tags 379
  - WScript statements 380
- simple data type 44
- sin (Math class) 169
- slice (Array class) 129
- slice (String class) 205
- SOAP 392
- sort (Array class) 130
- Source tag 331
- space characters 30
- splice (Array class) 133
- split (String class) 206
- sqrt (Math class) 169
- standalone WtBean 397
- standard master template 313
- start template 394
- statement 279, 284, 288, 297, 300, 301
- statement block 279
- String class 191
  - charAt 192
  - charCodeAt 193
  - concat 194
  - equals 195
  - fromCharCode 195
  - getClassName 196
  - indexOf 197
  - lastIndexOf 198
  - match 199
  - replace 201
  - search 203
  - setValue 204
  - slice 205
  - split 206
  - substr 207
  - substring 208
  - toLowerCase 209
  - toString 209
  - toUpperCase 210
  - valueOf 210
- string concatenation operator 72
- string data type 46

string literals 35  
String() 113  
stringlike 47  
strings see string literals  
style 393  
substr (String class) 207  
substring (String class) 208  
subtraction 65  
suffix.clt 309  
switch 288  
synchronized dialog 384, 393  
system access control 393  
system attributes  
    Java 340  
system exit  
    WTSleep 366  
system object 393

**T**  
TAC 395  
tag 393  
tan (Math class) 169  
TCP/IP 393  
template 394  
    class 383  
    example 27  
    master 394  
    object 394  
    start 394  
terminal application 394  
terminal hardcopy printing 394  
terminate  
    processing (Exit tag) 267  
test (RegExp class) 190  
text literals 33  
this 77  
Thread 386  
toGMTString (Date class) 145  
toLocaleString (Date class) 146  
toLowerCase (String class) 209  
toString (Array class) 134  
toString (Boolean class) 139  
toString (Date class) 146  
toString (host data object class) 155

toString (Number class) 172  
toString (Object class) 175  
toString (String class) 209  
toString (WT\_LdapConnection class) 254  
toUpperCase (String class) 210  
transaction 394  
transaction code/TAC 395  
tree structure  
    hierarchical (LDAP) 235  
    LDAP 235  
try block 305  
type conversion 47

## U

UDDI 395  
unbind (WT\_LdapConnection class) 254  
undefined data type 45  
unescape() 114  
Unicode 395  
unshift (Array class) 135  
update 284  
UPIC 395  
URI 395  
URL 395  
user exits 258, 335, 395  
    dynamic pages without host application 335  
user exits (C/C++) 350  
    defining 351  
    examples 353  
    link 351  
    supplied files 350  
user exits (Java)  
    activating Java support 338  
    environment variables 340  
    system attributes 340  
user exits (ready-made)  
    CheckLogin 357  
    CheckProcess 357  
    Creationtime 358  
    Delfile 359  
    FreeBuffer 359  
    FreeNameInPool 359  
    Getdate 360  
    Getdir 360

- Getfile [360](#)
- GetInstallDir [361](#)
- Gettime [361](#)
- LockNameInPool [362](#)
- Modificationtime [362](#)
- Putfile [363](#)
- ReleaseStationName [363](#)
- ReplaceByConfigFile [364](#)
- ReserveStationName [364](#)
- SendMail [365](#)
- user functions (WT\_Userexit class) [258](#)
- user ID [396](#)
- using your own functions [335](#)
- UTM see openUTM

## V

- value [295](#)
  - pass to function [300](#)
- value range of a data type [384](#)
- valueOf (Array class) [136](#)
- valueOf (Boolean class) [140](#)
- valueOf (Date class) [147](#)
- valueOf (Document class) [152](#)
- valueOf (host data object class) [156](#)
- valueOf (Number class) [172](#)
- valueOf (Object class) [179](#)
- valueOf (String class) [210](#)
- valueOf (WT\_LdapConnection class) [255](#)
- var [295](#)
- variable [43](#), [396](#)
  - data type [43](#)
  - for template [295](#)
  - global [295](#)
  - initialization [53](#)
  - lifetime [51](#)
  - local [295](#)
  - local and global [49](#)
  - overview of name spaces [52](#)
- variable declaration [277](#)
  - explicit [295](#)
  - implicit [295](#)
- variable type [295](#)
- visibility [396](#)

## W

- web server [396](#)
- web service [396](#)
- WebLab [23](#)
- WebTransactions
  - session [392](#)
- WebTransactions application [396](#)
- WebTransactions platform [396](#)
- WebTransactions server [396](#)
- while [281](#), [283](#)
- white spaces [30](#)
- with [277](#), [301](#)
- write (Document class) [153](#)
- writeln (Document class) [153](#)
- writeToTrace() [115](#)
- WSDL [396](#)
- WT\_Communication class [211](#)
  - close [212](#)
  - equals [212](#)
  - getClassName [213](#)
  - getModule [213](#)
  - open [214](#)
  - receive [214](#)
  - send [215](#)
- WT\_Filter class [216](#)
  - dataObjectToFormattedXML [219](#)
  - dataObjectToXML [217](#)
  - methodCallToXML [222](#)
  - objectTreeToXML [223](#)
  - XML\_SAXParse [227](#)
  - XMLToDataObject [224](#)
  - XMLToMethodCall [225](#)
  - XMLToObjectTree [226](#)
- WT\_LdapConnection class [234](#)
  - add [237](#)
  - bind [238](#)
  - bindSasl [239](#)
  - compare [240](#)
  - deleteEntry [241](#)
  - equals [241](#)
  - explodeDn [242](#)
  - firstEntry [243](#)
  - getClassName [244](#)
  - getDn [244](#)

- getEntries [245](#)
- getOption [246](#)
- modify [247](#)
- nextEntry [248](#)
- search [249](#)
- setOption [253](#)
- toString [254](#)
- unbind [254](#)
- valueOf [255](#)
- WT\_THIS [310](#)
- WT\_THIS operator [77](#)
- WT\_Userexit class [258](#)
  - user functions [258](#)
- WTBean [397](#)
- wtDataform [265](#)
- wtDoUntil [272](#)
- wtDoWhile [271](#)
- wtExit [267](#)
- wtIf [269](#)
- wtInclude [268](#)
- WTKernel.lib [351](#)
- WTML [23](#), [397](#)
- WTML tag [397](#)
- WTML tags
  - dynamic output [25](#)
  - short reference guide [379](#)
  - wtDataform [265](#)
  - wtDoUntil [272](#)
  - wtDoWhile [271](#)
  - wtExit [267](#)
  - wtIF [269](#)
  - wtInclude [268](#)
  - wtOnCreateScript [274](#)
  - wtOnReceiveScript [275](#)
  - wtRem [264](#)
- wtOnCreateScript [274](#)
- wtOnReceiveScript [275](#)
- WTPublic.h [350](#)
- wtRem [264](#)
- WTScript [26](#), [397](#)
- WTScript operator
  - using with Java objects [348](#)
- WTScript statements [277](#)
  - short reference guide [380](#)

- WTSleep [366](#)
- WTSystemExits.dl [350](#)
- WTSystemExits.so [350](#)
- WTUserexit.c [350](#)
- WTUserexits.dll [350](#)
- WTUserexits.so [350](#)
- WWW browser [382](#)
- WWW server [396](#)

## X

- XML [397](#)
- XML schema [397](#)
- XML\_SAXParse (WT\_Filter class) [227](#)
- XMLToDataObject (WT\_Filter class) [224](#)
- XMLToMethodCall (WT\_Filter class) [225](#)
- XMLToObjectTree (WT\_Filter class) [226](#)