



1
2
3
4
5

Document Number: DSP0221

Date: 2011-10-18

Version: 3.0.0a

6 **Managed Object Format (MOF)**

Information for Work-in-Progress version:

This document is subject to change at any time without further notice.

It expires on: 04/15/2012

Provide any comments through the DMTF Feedback Portal:

<http://www.dmtf.org/standards/feedback>

IMPORTANT: This specification is not a standard. It does not necessarily reflect the views of the DMTF or all of its members. Because this document is a Work in Progress, this specification may still change, perhaps profoundly. This document is available for public review and comment until the stated expiration date.

7

8 **Document Type: Specification**

9 **Document Status: Work in Progress - not a DMTF Standard**

10 **Document Language: en-US**

11

12 Copyright Notice

13 Copyright © 2011 Distributed Management Task Force, Inc. (DMTF). All rights reserved.

14 DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems
15 management and interoperability. Members and non-members may reproduce DMTF specifications and
16 documents, provided that correct attribution is given. As DMTF specifications may be revised from time to
17 time, the particular version and release date should always be noted.

18 Implementation of certain elements of this standard or proposed standard may be subject to third party
19 patent rights, including provisional patent rights (herein "patent rights"). DMTF makes no representations
20 to users of the standard as to the existence of such rights, and is not responsible to recognize, disclose,
21 or identify any or all such third party patent right, owners or claimants, nor for any incomplete or
22 inaccurate identification or disclosure of such rights, owners or claimants. DMTF shall have no liability to
23 any party, in any manner or circumstance, under any legal theory whatsoever, for failure to recognize,
24 disclose, or identify any such third party patent rights, or for such party's reliance on the standard or
25 incorporation thereof in its product, protocols or testing procedures. DMTF shall have no liability to any
26 party implementing such standard, whether such implementation is foreseeable or not, nor to any patent
27 owner or claimant, and shall have no liability or responsibility for costs or losses incurred if a standard is
28 withdrawn or modified after publication, and shall be indemnified and held harmless by any party
29 implementing the standard from any and all claims of infringement by a patent owner for such
30 implementations.

31 For information about patents held by third-parties which have notified the DMTF that, in their opinion,
32 such patent may relate to or impact implementations of DMTF standards, visit
33 <http://www.dmtf.org/about/policies/disclosures.php>.

34
35

36 Contents

37 Foreword 5

38 Introduction 6

39 Typographical Conventions 6

40 Deprecated Material 6

41 Experimental Material 6

42 1 Scope 7

43 2 Normative references 7

44 3 Terms and definitions 8

45 3.1 Managed object format 8

46 3.2 MOF specification file 8

47 3.3 MOF specification 9

48 4 Symbols and abbreviated terms 9

49 4.1 MOF 9

50 4.2 ABNF 9

51 5 MOF specifications 9

52 5.1 Encoding 9

53 5.2 Comments 9

54 5.3 Compiler directives 10

55 6 MOF language elements 11

56 6.1 Qualifier declaration 11

57 6.2 Type declarations 13

58 6.2.1 Enumeration declaration 13

59 6.2.2 Structure declaration 17

60 6.2.3 Class declaration 19

61 6.2.4 Association declaration 25

62 6.2.5 CIM Base types declarations 26

63 6.3 Instance and value definitions 28

64 Annex A (normative) MOF grammar description 31

65 A.1 MOF specification 32

66 A.2 Compiler directive 32

67 A.3 Structure declaration 32

68 A.4 Class declaration 33

69 A.5 Association declaration 33

70 A.6 Enumeration declaration 33

71 A.7 Instance declaration 34

72 A.8 Qualifier declaration 35

73 A.9 Qualifier list 36

74 A.10 Property declaration 36

75 A.11 Method declaration 36

76 A.12 Parameter declaration 37

77 A.13 Values 37

78 A.14 Base data types 38

79 A.15 Names 39

80 A.16 Base type values 39

81 Annex B (normative) MOF keywords 42

82 Annex C (informative) Example MOF specification 43

83 C.1 GOLF_Schema.mof 43

84 C.2 GOLF_Base.mof 43

85 C.3 GOLF_Club.mof 44

86 C.4 GOLF_ClubMember.mof 44

87 C.5 GOLF_Professional.mof 45

88 C.6 GOLF_Locker.mof 46

89	C.7	GOLF_MemberLocker.mof	46
90	C.8	GOLF_Lesson.mof	46
91	C.9	GOLF_Address.mof	47
92	C.10	GOLF_Date.mof	47
93	C.11	GOLF_PhoneNumber.mof	47
94	C.12	GOLF_ResultCodeEnum.mof	47
95	C.13	GOLF_MonthsEnum.enum	49
96	C.14	GOLF_StatesEnum.mof	49
97	C.15	JohnDoe.mof	50
98			

99 **Tables**

100	1)	Table 1 – Standard Compiler Directives.....	10
-----	----	---	----

101

102 Foreword

103 The CIM v3 Managed Object Format (MOF) (DSP0221) was prepared by the DMTF Architecture Working
104 Group.

105 Versions marked as "DMTF Standard" are approved standards of the Distributed Management Task
106 Force (DMTF).

107 DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems
108 management and interoperability. For information about the DMTF see <http://www.dmtf.org>.

109 Acknowledgments

110 The DMTF acknowledges the following individuals for their contributions to this document:

111 Editors:

- 112 • George Ericson – EMC
- 113 • Wojtek Kozaczynski – Microsoft

114 Contributors:

- 115 • Andreas Maier – IBM
- 116 • Jim Davis – WBEM Solutions
- 117 • Karl Schopmeyer – Inova Development
- 118 • Lawrence Lamers – VMware
- 119 • Mike Brasher – Microsoft

120 Introduction

121 This document specifies the DMTF Managed Object Format (MOF), which is a definition language used
122 to specify the interfaces of managed resources (storage, networking, compute, software) conformant with
123 the CIM metamodel defined in [DSP0004](#).

Comment [W.K.1]: Update the hyperlink to the final version

124 Typographical Conventions

125 The following typographical conventions are used in this document:

- 126 • Document titles are marked in *italics*.
- 127 • Important terms that are used for the first time are marked in *italics*.
- 128 • Examples are shown in the code blocks `code`.

129 Deprecated Material

130 Deprecated material is not recommended for use in new development efforts. Existing and new
131 implementations may use this material, but they shall move to the favored approach as soon as possible.
132 CIM services shall implement any deprecated elements as required by this document in order to achieve
133 backwards compatibility. Although CIM clients may use deprecated elements, they are directed to use the
134 favored elements instead.

135 Deprecated material should contain references to the last published version that included the deprecated
136 material as normative material and to a description of the favored approach.

137 The following typographical convention indicates deprecated material:

138 DEPRECATED

139 Deprecated material appears here.

140 DEPRECATED

141 In places where this typographical convention cannot be used (for example, tables or figures), the
142 "DEPRECATED" label is used alone.

143 Experimental Material

144 Experimental material has yet to receive sufficient review to satisfy the adoption requirements set forth by
145 the DMTF. Experimental material is included in this document as an aid to implementers who are
146 interested in likely future developments. Experimental material may change as implementation
147 experience is gained. It is likely that experimental material will be included in an upcoming revision of the
148 document. Until that time, experimental material is purely informational.

149 The following typographical convention indicates experimental material:

150 EXPERIMENTAL

151 Experimental material appears here.

152 EXPERIMENTAL

153 In places where this typographical convention cannot be used (for example, tables or figures), the
154 "EXPERIMENTAL" label is used alone.

155

156 1 Scope

157 This document describes the syntax, semantics and the use of the Managed Object Format (MOF) for the
158 DMTF Common Information Model (CIM) as defined in [DSP0004](#) version 3.0.

159 The MOF provides the means to specify interface definitions of managed object types; including their
160 properties, behavior and relationships with other objects. In the CIM context managed objects include
161 logical concepts like policies, as well as real-world resource such as disk drives, network endpoints or
162 software components.

163 MOF is used to define industry-standard object types, published by the DMTF as the CIM schema and
164 other schemas, as well as user/vendor-defined object types that may or may not be derived from object
165 types defined in schemas published by the DMTF.

166 This document does not describe specific CIM implementations, application programming interfaces
167 (APIs), or communication protocols.

168 2 Normative references

169 The following documents are indispensable for the application of this document. For dated or versioned
170 references, only the edition cited (including any corrigenda or DMTF update versions) applies. For
171 references without a date or version the latest published edition of the referenced document (including
172 any corrigenda or DMTF update versions) applies.

173 DMTF DSP0004, *Common Information Model (CIM) Infrastructure 3.0*

174 [http://members.dmtf.org/apps/org/workgroup/technical/dmtf-
175 arch/download.php/62435/DSP0004_3.0.0a_wipc2.docx](http://members.dmtf.org/apps/org/workgroup/technical/dmtf-arch/download.php/62435/DSP0004_3.0.0a_wipc2.docx)

176 IETF RFC5234, *Augmented BNF for Syntax Specifications: ABNF*, January 2008

177 <http://tools.ietf.org/html/rfc5234>

178 ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards

179 <http://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtype>

180 ISO 639-1:2002, *Codes for the representation of names of languages – Part 1: Alpha-2 code*

181 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22109

182 ISO 639-2:1998, *Codes for the representation of names of languages – Part 2: Alpha-3 code*

183 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=4767

184 ISO 639-3:2007, *Codes for the representation of names of languages – Part 3: Alpha-3 code for
185 comprehensive coverage of languages*

186 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39534

187 ISO 3166-1:2006, *Codes for the representation of names of countries and their subdivisions – Part 1:
188 Country codes*

189 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39719

190 ISO 3166-2:2007, *Codes for the representation of names of countries and their subdivisions – Part 2:
191 Country subdivision code*

192 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39718

193 ISO 3166-3:1999, *Codes for the representation of names of countries and their subdivisions – Part 3:
194 Code for formerly used names of countries*

195 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=2130

- 196 ISO/IEC 10646:2003, *Information technology – Universal Multiple-Octet Coded Character Set (UCS)*
197 [http://standards.iso.org/ittf/PubliclyAvailableStandards/c039921_ISO_IEC_10646_2003\(E\).zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c039921_ISO_IEC_10646_2003(E).zip)
- 198 ISO/IEC 10646:2003/Amd 1:2005, *Information technology -- Universal Multiple-Octet Coded Character*
199 *Set (UCS) -- Amendment 1: Glagolitic, Coptic, Georgian and other characters*
200 [http://standards.iso.org/ittf/PubliclyAvailableStandards/c040755_ISO_IEC_10646_2003_Amd_1_2005\(E\).](http://standards.iso.org/ittf/PubliclyAvailableStandards/c040755_ISO_IEC_10646_2003_Amd_1_2005(E).zip)
201 [zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c040755_ISO_IEC_10646_2003_Amd_1_2005(E).zip)
- 202 ISO/IEC 10646:2003/Amd 2:2006, *Information technology -- Universal Multiple-Octet Coded Character*
203 *Set (UCS) -- Amendment 2: N'Ko, Phags-pa, Phoenician and other characters*
204 [http://standards.iso.org/ittf/PubliclyAvailableStandards/c041419_ISO_IEC_10646_2003_Amd_2_2006\(E\).](http://standards.iso.org/ittf/PubliclyAvailableStandards/c041419_ISO_IEC_10646_2003_Amd_2_2006(E).zip)
205 [zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c041419_ISO_IEC_10646_2003_Amd_2_2006(E).zip)
- 206 ISO/IEC 14750:1999, *Information technology – Open Distributed Processing – Interface Definition*
207 *Language*
208 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=25486
- 209 OMG, *Object Constraint Language, Version 2.0*
210 <http://www.omg.org/cgi-bin/doc?formal/2006-05-01>
- 211 OMG, *UML Superstructure Specification, Version 2.1.1*
212 <http://www.omg.org/cgi-bin/doc?formal/07-02-05>
- 213 The Unicode Consortium, *The Unicode Standard, Version 5.2.0, Annex #15: Unicode Normalization*
214 *Forms*
215 <http://www.unicode.org/reports/tr15/>

216 **3 Terms and definitions**

217 Some terms used in this document have a specific meaning beyond the common English meaning. Those
218 terms are defined in this clause.

219 The terms "shall" ("required"), "shall not," "should" ("recommended"), "should not" ("not recommended"),
220 "may," "need not" ("not required"), "can" and "cannot" in this document are to be interpreted as described
221 in ISO/IEC Directives, Part 2, Annex H. The terms in parenthesis are alternatives for the preceding terms,
222 for use in exceptional cases when the preceding term cannot be used for linguistic reasons. Note that
223 ISO/IEC Directives, Part 2, Annex H specifies additional alternatives. Occurrences of such additional
224 alternatives shall be interpreted in their normal English meaning.

225 The terms "clause," "subclause," "paragraph," and "annex" in this document are to be interpreted as
226 described in ISO/IEC Directives, Part 2, Clause 5.

227 The terms "normative" and "informative" in this document are to be interpreted as described in ISO/IEC
228 Directives, Part 2, Clause 3. In this document, clauses, subclauses, or annexes labeled "(informative)" do
229 not contain normative content. Notes and examples are always informative elements.

230 The Terms and Definitions defined in DSP0004 are included by reference.

231 **3.1 Managed object format**

232 Refers to the language described in this specification.

233 **3.2 MOF file**

234 Refers to a document with the content that conforms to the MOF syntax described by this specification.

235 **3.3 MOF compilation unit**

236 Refers to a set of MOF files, which includes those files that are explicitly listed as the input to the MOF
237 compiler and the files directly or transitively included from the input files using the include pragma
238 compiler directive.

239 **3.4 MOF compiler**

240 Program, which transforms a MOF compilation unit into another representation, such as an AST.

241 **4 Symbols and abbreviated terms**

242 The Symbols and Abbreviations defined in DSP0004 are included by reference.

243 **4.1 MOF**

244 Managed Object Format

245 **4.2 ABNF**

246 Augmented BNF

247 **4.3 IDL**

248 Interface definition language (http://en.wikipedia.org/wiki/Interface_description_language)

249 **5 MOF file content**

250 A MOF file contains MOF language statements, compiler directives and comments.

251 **5.1 Encoding**

252 The content of a MOF file shall be represented in Normalization Form C
253 (<http://www.unicode.org/reports/tr15/>) and in the coded representation form UTF-8 (defined in ISO/IEC
254 10646:2003).

255 The content represented in UTF-8 should not have a signature sequence (EF BB BF, as defined in Annex
256 H of ISO/IEC 10646:2003).

257 **5.2 White space**

258 White space in a MOF file is any combination of the following characters: space (U+0020), carriage return
259 (U+000D), line feed (U+000A).

260 **5.3 Line termination**

261 The end of a line in a MOF file is indicated by one of the following:

- 262 • A sequence of carriage return (U+000D) followed by line feed (U+000A)
- 263 • A line feed (U+000A)
- 264 • Implicitly by the end of the MOF specification file, if the line is not ended by line end
265 characters.

266 The different line end characters may be arbitrarily mixed within a single MOF file.

267 **5.4 Comments**

268 Comments in a MOF file do not create, modify, or annotate language elements. They shall be treated as
 269 white space by the MOF compilers and therefore are not represented in the language grammar.

270 Comments may appear anywhere in MOF syntax where white space is allowed and are indicated by
 271 either a leading double slash (//) or a pair of matching /* and */ character sequences.

272 A // comment is terminated by the end of line (see 5.3).

```
273 uint16 MyProperty; // This is an example comment
```

274 A /* comment is terminated by the next */ sequence or by the end of the MOF file (whichever comes
 275 first).

```
276 uint16 MyProperty; /* This is also  

    277 a comment */
```

278 **5.5 Compiler directives**

279 Compiler directives direct the processing of MOF files. Compiler directives do not create, modify, or
 280 annotate the language elements.

281 Compiler directives shall conform to the following structure

```
282 compiler Directive = "#pragma" directiveName "(" stringValue ")"  

    283 directiveName = IDENTIFIER
```

284 where IDENTIFIER is defined in Annex A, A.15.

285 The current standard compiler directives are listed in [Table 1](#).

286 **Table 1 – Standard Compiler Directives**

Compiler Directive	Description
#pragma include(<path>)	<p>This directive specifies that the referenced MOF specification file is included. The content of the referenced file shall be textually inserted at the point where the include directive is encountered.</p> <p>The path can be either an absolute file system path, or a relative path. If the path is relative, it is relative to the directory of the file with this the pragma.</p> <p>The content of the included field shall be one or more complete syntactic MOF production as defined in Annex A, A.1.</p>

Comment [W.K.2]: Should we keep the absolute path.

 What is the format of the relative path? Should it start with "." or just the name of sub-directory followed by "\"

287 A MOF compiler may support additional compiler directives. Such new compiler directives are referred to
 288 as *vendor-specific compiler directives*. Vendor-specific compiler directives should have names that are
 289 unlikely to collide with the names of standard compiler directives defined in future versions of this
 290 specification. Future versions of this specification will not define compiler directives with names that begin
 291 with underscore (_, U+005F). Therefore, it is recommended that the names of vendor-specific compiler
 292 directives conform to the following structure:

```
293 directiveName = "_" org-id "_" IDENTIFIER
```

294 where org-id includes a copyrighted, trademarked, or otherwise unique name owned by the business
 295 entity that defines the compiler directive or that is a registered ID assigned to the business entity by a
 296 recognized global authority.

297 Vendor-specific compiler directives that are not understood by a MOF compiler shall be reported, but
 298 should be ignored by default. Thus, the use of vendor-specific compiler directives may affect the
 299 interoperability of MOF.

300 **6 MOF language elements**

301 MOF is an interface definition language (IDL) that is implementation language independent, and has
 302 syntax that should be familiar to programmers that have worked with other IDLs.

303 A MOF specification includes the following kinds of elements

- 304 • Compiler directives that direct the processing of the specification
- 305 • Qualifier declarations
- 306 • Type declarations such as classes, structures or enumerations, and
- 307 • Instances declarations.

308 Elements of MOF language are introduced and exemplified one-by-one in a sequence that progressively
 309 builds a meaningful MOF specification. To make the examples consistent we have created a small
 310 specification (a model) of a golf club membership. The root elements in the model are defined in the
 311 GOLF schema. All files of the model are listed in [ANNEX C](#).

312 A complete description of the MOF syntax is provided in [ANNEX A](#).

313 **6.1 Qualifiers**

314 A qualifier is a named and typed meta-data element. The value of a qualifier is associated with and
 315 describes a schema element. A detailed discussion of the qualifier concept and the list of standard
 316 qualifiers can be found in Section 7 of DSP0004.

317 A qualifier declaration in MOF corresponds to the QualifierType meta-model element, and is defined by
 318 the following ABNF rules:

```

319 qualifierDeclaration = [ qualifierList ] [TRANSLATABLE]
320                     QUALIFIER qualifierName ":" qualifierType qualifierScope [qualifierPolicy] ":"
321 qualifierName       = IDENTIFIER
322 qualifierType       = singleValueQualifierType / arrayValueQualifierType
323 singleValueQualifierType = ( primitiveType / enumName ) [ "=" defaultQualifierValue ]
324 arrayValueQualifierType = primitiveType "[" "]" [ "=" defaultQualifierValueList ]
325 defaultQualifierValue = literalValue
326 defaultQualifierValueList = "{" primitiveTypeValueList "}"
327 primitiveTypeValueList = literalPrimitiveTypeValue *( "," literalValue)
328 qualifierScope       = "," SCOPE "(" ANY / qualifiedElementList ")"
329 qualifierPolicy      = "," POLICY "(" APPLYMANY / APPLYONCE / RESTRICTED ")"
330 qualifiedElementList = qualifiedElement *( "," qualifiedElement )
331 qualifiedElement     = STRUCTURE / CLASS / ASSOCIATION /
332                     ENUMERATION / ENUMERATIONLITERAL /
    
```

Comment [W.K.3]: What is the default value if the policy is not specified? ApplyMany?

```

333     PROPERTY / REFPROPERTY /
334     METHOD / PARAMETER /
335     QUALIFIER
336 TRANSLATABLE      = "translatable"      ; keyword: case insensitive
337 SCOPE             = "scope"             ; keyword: case insensitive
338 ANY               = "any"               ; keyword: case insensitive
339 POLICY            = "policy"            ; keyword: case insensitive
340 APPLYMANY        = "applymany"         ; keyword: case insensitive
341 APPLYONCE        = "applyonce"         ; keyword: case insensitive
342 RESTRICTED       = "restricted"        ; keyword: case insensitive
343 ENUMERATIONLITERAL = "enumerationliteral" ; keyword: case insensitive
344 PROPERTY         = "property"          ; keyword: case insensitive
345 REFPROPETY       = "reference"          ; keyword: case insensitive
346 METHOD            = "method"            ; keyword: case insensitive
347 PARAMETER        = "parameter"         ; keyword: case insensitive
348 QUALIFIER        = "qualifier"         ; keyword: case insensitive

```

349

350 The following MOF fragment defines the qualifier AggregationKind. The AggregationKind qualifier specifies
 351 the literals that define the kind of aggregation for a property with type reference. The type of the qualifier
 352 is a string enumeration with three literals; None, Shared and Composite.

```

353
354 Qualifier AggregationKind : CIM_AggregationKindEnum = None,
355     Scope(reference), Policy (applyMany);
356
357 enumeration CIM_AggregationKindEnum : string {
358     None,
359     Shared,
360     Composite
361 };

```

Comment [W.K.4]: Should we follow the meta-model naming convention and spell the enum literals starting with lower-case letters?

362

363 A qualifier value in MOF represents an instance of the Qualifier meta-model element defined in DSP0004.
 364 The set of qualifier values specified in MOF on a schema element shall conform to the following
 365 qualifierList ABNF rule:

```

366 qualifierList      = "[" qualifier *( "," qualifier ) "]"
367 qualifier          = qualifierName [ qualifierParameter ]
368 qualifierName      = IDENTIFIER
369 qualifierParameter = qualifierValueInitializer / qualifierValueArrayInitializer
370 qualifierValueInitializer = "(" literalPrimitiveTypeValue ")"
371 qualifierValueArrayInitializer = "{" primitiveTypeValue *( "," primitiveTypeValue ) "}"

```

372

373 The list of qualifier scopes (see the qualifiedElement rule above) includes the scope "qualifier", which
 374 implies that qualifier declarations can be qualified. The list of standard qualifiers is defined in DSP0004.
 375 Currently there are two standard qualifiers that can be specified on a qualifier declaration: Description and
 376 Deprecated. Note that there may be vendor-specific qualifiers with scope "qualifier".

377 6.2 Types

378 The data types in CIM are:

- 379 • Enumerations
- 380 • Structures
- 381 • Classes
- 382 • Associations, and
- 383 • Primitive types.

384 The subclauses below define how enumerations, structures, classes and associations are declared in
 385 MOF, and what primitive types are available in MOF.

Comment [W.K.5]: Changed "base type" to "primitive type" to make it consistent with the meta-model.

386 6.2.1 Enumeration declaration

387 There are two kinds of enumerations in CIM

- 388 • Integer enumerations, and
- 389 • String enumerations.

390 CIM integer enumerations are comparable to enumerations in programming languages; each
 391 enumeration literal in an integer enumeration is represented by a distinct integer value.

392 In string enumerations, which can be found in UML and are similar to XML enumerations, each
 393 enumeration literal is represented by a distinct string value.

394 An enumeration declaration in MOF corresponds to the Enumeration meta-model element defined in
 395 DSP0004 and shall conform to the following enumDeclaration ABNF rule:

```

396 enumDeclaration      = [ qualifierList ] [FINAL]
397                      ENUMERATION enumName ":"
398                      ( integerEnumDeclaration /
399                      stringEnumDeclaration /
400                      derivedEnumDeclaration )
401 enumName              = ( IDENTIFIER / schemaQualifiedName )

```

```

402 integerEnumDeclaration = DT_UnsignedInteger "{" integerEnumElement *(" " integerEnumElement) "}" ";
403 stringEnumDeclaration = DT_STRING "{" stringEnumElement *(" " stringEnumElement) "}" ";
404 integerEnumElement = [ qualifierList ] enumLiteral ["=" integerValue];
405 stringEnumElement = [ qualifierList ] enumLiteral ["=" stringValue];
406 derivedEnumDeclaration = ( IDENTIFIER / schemaQualifiedName )
407     "{" enumElement *(" " enumElement) "}" ";
408 enumElement = enumLiteral ["=" (integerValue / stringValue) ] ";
409 enumLiteral = IDENTIFIER
410 ENUMERATION = "enumeration" ; keyword: case insensitive

```

Comment [W.K.6]: Modeled the values of integer enumerations non-optional.

Comment [W.K.7]: Left string values optional.

411

412 The integerEnumElement rule states that integer enumeration elements must have explicit integer values.
 413 Those values must be unique and ascending. The stringEnumElement rule states that the values of string
 414 enumeration elements are optional. If not declared, the value of a string enumeration element is assumed
 415 to be the same as its literal. This is illustrated in the examples below.

416 The golf club model contains a number of enumeration declarations. Enumerations can be defined at
 417 the schema level or inside declarations of structures, classes or associations. Enumerations defined inside
 418 those other types are referred to as "embedded" enumeration declarations. The names of schema-level
 419 enumerations must conform to the schemaQualifiedMOF rule that is, their names must start with the name
 420 of the schema followed by the underscore (U+005F).

421 The schema-level string enumeration GOLF_MonthsEnum shown below defines months of the year.

422

```

423 enumeration GOLF_MonthsEnum : string {
424     January,
425     February,
426     March,
427     April,
428     May,
429     June,
430     July,
431     August,
432     September,
433     October,
434     November,
435     December
436 };

```

437 As explained above, string enumerations don't require assigning values to their literals. If a value is not
 438 assigned to a literal, it is assumed that it is identical to the literal itself, so for example the value of the literal
 439 January above is "January".

440 The GOLF_StatesEnum is an example of another schema-level string enumeration that assigns values to
 441 all of its literals, which are different than the literal names.

442

```

443 enumeration GOLF_StatesEnum : string {

```

```
444     AL = "Alabama",
445     AK = "Alaska",
446     AZ = "Arizona",
447     AR = "Arkansas",
448     CA = "California",
449     CO = "Colorado",
450     CT = "Connecticut",
451     DE = "Delaware",
452     FL = "Florida",
453     GA = "Georgia",
454     HI = "Hawaii",
455     ID = "Idaho",
456     IL = "Illinois",
457     IN = "Indiana",
458     IA = "Iowa",
459     KS = "Kansas",
460     LA = "Louisiana",
461     ME = "Maine",
462     MD = "Maryland",
463     MA = "Massachusetts",
464     MI = "Michigan",
465     MS = "Mississippi",
466     MO = "Missouri",
467     MT = "Montana",
468     NE = "Nebraska",
469     NV = "Nevada",
470     NH = "New Hampshire",
471     NJ = "New Jersey",
472     NM = "New Mexico",
473     NY = "New York",
474     NC = "North Carolina",
475     ND = "North Dakota",
476     OH = "Ohio",
477     OK = "Oklahoma",
478     OR = "Oregon",
479     PA = "Pennsylvania",
480     RI = "Rhode Island",
481     SC = "South Carolina",
482     SD = "South Dakota",
483     TX = "Texas",
484     UT = "Utah",
485     VT = "Vermont",
486     VA = "Virginia",
487     WA = "Washington",
488     WV = "West Virginia",
489     WI = "Wisconsin",
490     WY = "Wyoming"
491 };
```

492 The following two enumerations are examples of embedded integer enumerations that derive from each
 493 other. The declaration of the MemberStatusEnum is embedded in the declaration of the GOLF_ClubMember
 494 class. It defines three elements and assigns integer values to them. The assignment of values to the
 495 integer enumeration literals is not optional (see the integerEnumElement rule). The values shall to be
 496 assigned in ascending order, but don't have to be consecutive.

```

497
498 [Description (
499     "Instances of this class represent members of a golf club." )]
500 class GOLF_ClubMember: GOLF_Base {
501
502 // ===== embedded enumerations =====
503     enumeration MemberStatusEnum : Uint16 {
504         Basic = 0,
505         Extended = 1,
506         VP = 2
507     };
508
509 // ===== properties =====
510     MemberStatusEnum Status;
511     GOLF_Date MembershipEstablishedDate;
512
513     ...
514 };
    
```

515 The ProfessionalStatusEnum enumeration is defined in the GOLF_Professional class that inherits from the
 516 GOLF_ClubMember class. It derives from the MemberStatusEnum enumeration declared in the parent class
 517 and adds two elements to it. The elements are not given consecutive values (the values of the
 518 MemberStatusEnum end at 2 and the values of ProfessionalStatusEnum start at 6). This is intent ended to
 519 leave the range 3, 4 and 5 for elements that can be added to the MemberStatusEnum enumeration at later
 520 time.

```

521
522 class GOLF_Professional : GOLF_ClubMember {
523 // ===== embedded structures =====
524     ...
525
526 // ===== embedded enumerations =====
527     enumeration ProfessionalStatusEnum : MemberStatusEnum {
528         Professional = 6,
529         SponsoredProfessional = 7
530     };
531
532 // ===== properties =====
533     ...
534 };
    
```

535
 536 Declarations of CIM enumerations should meet the following model integrity constraints
 537 • Only qualifiers with the scope “enumeration” can be applied to enumeration declarations

Comment [W.K.8]: Not the final list

- 538 • A derived enumeration can only add literals
- 539 • Enumeration literals shall be unique in an enumeration
- 540 • A derived enumeration is of the same primitive type as its base enumeration and therefore the
- 541 values assigned to its literals shall be of that type
- 542 • The elements in a derived enumeration of integer primitive type shall have values
- 543 • The definition order of *integer* enumeration literals is significant and their values shall increase
- 544 • The definition order of the *string* enumeration elements does not matter
- 545 • ...
- 546

547 6.2.2 Structure declaration

548 A CIM structure defines a complex type that has no independent identity, but can be used as a type of a
549 property, a method result or a method parameter.

550 The structure declaration in MOF corresponds to the Structure meta-model element element defined in
551 DSP0004 and shall conform to the structureDeclaration ABNF rule:.

```

552 structureDeclaration = [ qualifierList ] [FINAL]
553                     STRUCTURE structureName [superStructure]
554                     "{" *propertyDeclaration "}" ";"
555 structureName       = ( IDENTIFIER / schemaQualifiedName )
556 superStructure      = ":" ( IDENTIFIER / schemaQualifiedName )
557 FINAL               = "final" ; keyword: case insensitive
558 STRUCTURE           = "structure" ; keyword: case insensitive

```

559

560 Structure is a, possibly empty, collection of properties that can derive from another structure (see the
561 *general* association in CIM meta-model in DSP004) . A structure can be declared at the schema level,
562 and therefore be visible to all other structures, classes and associations in the schema, or its declaration
563 can be embedded in a class declaration and be visible only to that class and its ancestors.

564 A property declaration in MOF (for use by structures, classes and associations) corresponds to the
565 Property meta-model element defined in DSP0004 and shall conform to the propertyDeclaration ABNF
566 rule:

```

567 propertyDeclaration = [ qualifierList ] [KEY] [
568                     ( basePropertyDeclaration /
569                     structurePropertyDeclaration /
570                     enumPropertyDeclaration /
571                     classPropertyDeclaration /
572                     referecePropertyDeclaration )
573 basePropertyDeclaration = primitiveType propertyName [ array ] [ "=" primitiveTypeDefaultValue ] ";"

```

Comment [W.K.9]: The meta-model introduces two meta-properties of properties
- key, and
- static
I have only added the key, and would like to discuss the static on separate thread.

```

574 structurePropertyDeclaration = structureName propertyName [ array ] [ "=" structureDefaultValue ] ";"
575 enumPropertyDeclaration    = enumName propertyName [ array ] [ "=" enumDefaultValue ] ";"
576 classPropertyDeclaration   = className propertyName [ array ] [ "=" structureDefaultValue ] ";"
577 referencePropertyDeclaration = className REF propertyName [ array ] [ "=" referenceDefaultValue ] ";"
578 array                       = "[" "]"
579 propertyName                = IDENTIFIER
580 KEY                         = "key" ; keyword: case insensitive
581 REF                         = "ref" ; keyword: case insensitive

```

Comment [W.K.10]: We have removed the array size. Array size constraints should be specified using the Constraint qualifier.

582

583 The GOLF_Date is an example of a schema-level structure with three properties.

584

```

585 structure GOLF_Date {
586     UInt16 Year = 2000;
587     GOLF_MonthsEnum Month = GOLF_MonthsEnum.January;
588     [MinValue(1), MaxValue(31)]
589     UInt16 Day = 1;
590 }

```

591

592 All of the properties have default values that set the structure value to January 1, 2000 if no other values
593 are provided. The default value of the Month property can be simplified to the form

594

```
595 GOLF_MonthsEnum Month = January
```

596 as the enumeration type is implied by the property declaration. The Day property has the min and max
597 values defined using the MinValue and MaxValue qualifiers.

598 The use of the GOLF_Date structure as the type of a property is shown in previous section in the
599 declaration of the GOLF_ClubMember class; the property is called MembershipEstablishedDate.

600 The Sponsor structure, see below, is defined inside the GOLF_Professional class. It is an example of an
601 embedded structure declaration, which can be used only in the class it is defined in, or a derived class.

602

```

603 class GOLF_Professional : GOLF_ClubMember {
604 // ===== embedded structures =====
605     structure Sponsor {
606         string Name,
607         GOLF_Date ContractSignedDate;
608         Real32 ContractAmount;
609     };
610 // ===== embedded enumerations =====
611

```

```

612 enumeration ProfessionalStatusEnum : MemberStatusEnum {
613     Professional = 6,
614     SponsoredProfessional = 7
615 };
616
617 // ===== properties =====
618     [Override]
619     ProfessionalStatus Status = Professional;
620     Sponsor Sponsors[];
621     Boolean Ranked;
622 };
    
```

623 The declaration of the default value of the Status property is an example of short version of the
 624 ProfessionalStatus.Professional enumeration element value.

Comment [W.K.11]: Not the final list

625 A declaration of a CIM structure should conform to the following model integrity constraints

- 626 • Only qualifiers with the scope "structure" shall be applied to a structure declaration
- 627 • An embedded structure can inherit from a schema-level structure, but a schema-level structure shall not
 628 inherit from an embedded structure
- 629 • Property names in a structure shall be unique
- 630 • Overriding property shall not change the property type
- 631 • A structure shall not have a key property (a property with the modifier "key")
- 632 • Only qualifiers with the scope "property" shall be applied to property declaration
- 633 • Scalar properties should have single default value
- 634 • Array properties should have array default values, even if that array contains only a single element
- 635 •
- 636 • ...

637 **6.2.3 Class declaration**

638 A class defines both properties and methods (the behavior) of its instances, which have unique identity in
 639 the scope of a server, a namespace, and the class. A class may define embedded structures and
 640 enumerations.

641 A class declaration in MOF corresponds to the Class meta-model element defined in DSP0004 and shall
 642 conform to the classDeclaration ABNF rule:

```

643 classDeclaration = [ qualifierList ] *2classModifier
644                  [ABSTRACT] CLASS className [ superClass ]
645                  "{" *classFeature "}" ";"
646 classModifier   = ABSTRACT / FINAL
647 className       = schemaQualifiedName
648 superClass      = ":" schemaQualifiedName
649 classFeature    = propertyDeclaration /
650                  enumerationDeclaration /
    
```

Comment [W.K.12]: This rule allows the write the class modifiers in any order; abstract final, or final abstract. Should we change require a fixed order?

```

651         structureDeclaration /
652         methodDeclaration
653 ABSTRACT         = "abstract"           ; keyword: case insensitive
654 CLASS           = "class"             ; keyword: case insensitive

```

655

656 The embedded enumeration and structure declarations have been introduced in sections 6.2.1 and 6.2.2
657 respectively. Section 6.2.2 also introduced the property declaration.

658 A method declaration within a class declaration in MOF corresponds to the Method meta-model element
659 defined in DSP0004 and shall conform to the methodDeclaration ABNF rule:

```

660 methodDeclaration = [ qualifierList ] [ STATIC ] ( dataType [ array ] / VOID ) methodName
661                  "(" [ parameterList ] ")" ";"
662 methodName       = IDENTIFIER
663 VOID             = "void"             ; keyword: case insensitive
664 STATIC          = "static"           ; keyword: case insensitive
665 parameterList    = parameterDeclaration *( "," parameterDeclaration )

```

666

```

667 parameterDeclaration = [ qualifierList ] [ IN / OUT / INOUT ]
668                     ( primitiveParamDeclaration /
669                     structureParamDeclaration /
670                     enumParamDeclaration /
671                     classParamDeclaration /
672                     referenceParamDeclaration )
673 primitiveParamDeclaration = primitiveType parameterName [ array ] [ "=" primitiveTypeDefaultValue ]
674 structureParamDeclaration = structureName parameterName [ array ] [ "=" structureDefaultValue ]
675 enumParamDeclaration      = enumName parameterName [ array ] [ "=" enumDefaultValue ]
676 classParamDeclaration     = className parameterName [ array ] [ "=" structureDefaultValue ]
677 referenceParamDeclaration = className "REF" parameterName [ array ] [ "=" referenceDefaultValue ]
678 parameterName            = IDENTIFIER
679 IN                       = "in"             ; keyword: case insensitive
680 OUT                      = "out"            ; keyword: case insensitive
681 INOUT                   = "inout"          ; keyword: case insensitive

```

682

683 A class may define two kinds of methods

- 684 • Instance methods, which are invoked on an instance of the class (the concept is similar to the
- 685 "this" method argument in dynamic programming languages), and
- 686 • Static methods, which are invoked on the class.

687 The following intrinsic methods are predefined on each class, that is, the user does not have to specify
 688 them in MOF but the class provider should implement them:

- 689 • Intrinsic instance methods
 - 690 • Get
 - 691 • Delete
 - 692 • Modify
- 693 • Intrinsic static methods
 - 694 • Enumerate
 - 695 • Create.

696 A class can derive from another class, in which case it inherits the enumerations, structures, properties
 697 and methods of its superclass. A class can also derive from a structure, in which case it only inherits the
 698 properties of the parent structure.

699

700 Below are the declarations of three central classes of our GOLF schema example; GOLF_Base,
 701 GOLF_Club and GOLF_ClubMember.

702

```

703 // =====
704 // GOLF_Base
705 // =====
706 abstract class GOLF_Base {
707     [Description (
708         "InstanceID is a property that opaquely and uniquely identifies "
709         "an instance of a class that derives from the GOLF_Base structure. ")]
710     key string InstanceID;
711
712     [Description (
713         "A a short textual description (one- line string) of the "
714         "instance." ),
715         MaxLen (64)]
716     string Caption;
717 };
718
719 // =====
720 // GOLF_Club
721 // =====
722 [Description (
723     "Instances of this class represent golf clubs. A golf club is an "
724     "an organization that provides memembr services to golf players "
725     "both amateur and professional." )]
726 class GOLF_Club: GOLF_Base {
727 // ===== properties =====

```

Comment [W.K.13]: Replaced [Key] qualifier with *key* keyword.

```

728     string ClubName;
729     GOLF_Date YearEstablished;
730
731     GOLF_Address ClubAddress;
732     GOLF_Phone ClubPhoneNo;
733     GOLF_Fax ClubFaxNo;
734     string ClubWebSiteURL;
735
736     GOLF_ClubMember REF AllMembers[];
737     GOLF_Professional REF Professionals[];
738
739 // ===== methods =====
740     static GOLF_ResultCodeEnum AddNonProfessionalMember (
741         in GOLF_ClubMember newMember
742     );
743
744     static GOLF_ResultCodeEnum AddProfessional (
745         in GOLF_Professional newProfessional
746     );
747
748     static UInt32 GetMembersWithOutstandingFees (
749         in GOLF_Date referenceDate,
750         out GOLF_ClubMember REF lateMembers[]
751     );
752
753     static GOLF_ResultCodeEnum TerminateMembership (
754         in GOLF_ClubMember REF member
755     );
756 };
757
758 // =====
759 // GOLF_ClubMember
760 // =====
761 [Description (
762     "Intsances of this class represent members of a golf club." )]
763 class GOLF_ClubMember: GOLF_Base {
764
765 // ===== embeded enumerations =====
766     enumeration MemberStatusEnum : UInt16 {
767         Basic = 0,
768         Extended = 1,
769         VP = 2
770     };
771
772 // ===== properties =====
773     string FirstName;
774     string LastName;
775     MemberStatusEnum Status;

```

```

776     GOLF_Date MembershipEstablishedDate;
777
778     Real32 MembershipSignUpFee;
779     Real32 MonthlyFee;
780     GOLF_Date LastPaymentDate;
781
782     GOLF_Address MemberAddress;
783     GOLF_PhoneNumber MemberPhoneNo;
784     string MemberEmailAddress;
785
786 // ===== methods =====
787     GOLF_ResultCodeEnum SendPaymentReminderMessage();
788 };
    
```

789 The GOLF_Base class is the base from which all other GOLF schema classes derive. It is an abstract class
 790 and it introduces a key property. A key property should be of type string, although other primitive types can be
 791 used, and must have the key modifier keyword. The key property is used by class implementations in
 792 servers to uniquely identify instances of the class in a namespace. The clients should make no
 793 assumptions about its content when instance paths are returned by the server. In other words the class
 794 implementation in the server sets the key property to a value that allows it to uniquely identify the
 795 represented managed object when it gets the instance path back from the client. CIM v3 makes an
 796 assumption that clients should not construct instance paths and only use the paths returned by the
 797 server.

Comment [W.K.14]: This deserves a separate discussion in the meta-model document.

799 All methods in the GOLF schema classes return the same result type, which is the enumeration
 800 GOLF_ResultCodeEnum defining the DMTF stand result codes (see Annex C for the definition).

801
 802 All methods of the GOLF_Club class are static and three of them manage two light-weight associations
 803 represented by the properties AllMembers and Professionals. Those two properties are instance reference
 804 arrays and model one-way associations from the GOLF_Club to GOLF_ClubMember and GOLF_Professional
 805 respectively. The static methods perform the following operations.

AddNonProfessionalMember	<p>Adds new non-professional member to the club and adds that member to the AllMembers association.</p> <p>Notice that the design of the method suggests that the implementation of the method will have to call CreateInstance intrinsic method on the GOLF_ClubMember class to create a new member instance and get a reference to it.</p>
AddProfessional	<p>Adds new professional member to the club and adds that member to both the Professionals and the AllMembers associations.</p> <p>As above, the design implies that it calls CreateInstance intrinsic method on the GOLF_Professional class to create an instance of the professional member.</p>
TerminateMembership	<p>Terminates the club membership and removes the member from AllMembers and possibly Professionals associations, depending on the membership status.</p> <p>Notice that in this case the method gets a reference to the existing member instance as an input parameter.</p>

GetMembersWithOutstandingFees	<p>Finds club members that have not paid the club dues on time.</p> <p>This method returns references to the "late" members in the form of a reference array.</p>
-------------------------------	---

807

808 The GOLF_ClubMember class has one extrinsic instance method SendPaymentReminderMessage. The
 809 method has no explicit parameters as it takes an instance of GOLF_ClubMember as its default argument.
 810 The design suggests that this method is called on all members that are returned by the method
 811 GetMembersWithOutstandingFees described above.

812

813 CIM v3 introduces the ability to define default values for method parameters (see the
 814 primitiveParamDeclaration, structureParamDeclaration, enumParamDeclaration, classParamDeclaration and
 815 referenceParamDeclaration MOF grammar rules). An example is the GetNumberOfProfessionals method in the
 816 GOLF_Professional class.

817

```

818 // =====
819 // GOLF_Professional
820 // =====
821 class GOLF_Professional : GOLF_ClubMember {
822 // ===== embeded structures =====
823     structure Sponsor {
824         string Name,
825         GOLF_Date ContractSignedDate;
826         Real32 ContractAmount;
827     };
828
829 // ===== embeded enumerations =====
830     enumeration ProfessionalStatusEnum : MemberStatusEnum {
831         Professional = 6,
832         SponsoredProfessional = 7
833     };
834
835 // ===== properties =====
836     [Override]
837     ProfessionalStatusEnum Status = Professional;
838     Sponsor Sponsors[];
839     Boolean Ranked;
840
841 // ===== methods =====
842     static GOLF_ResultCodeEnum GetNumberOfProfessionals (
843         out UInt32 NoOfPros,
844         in ProfessionalStatusEnum = Professional
845     )
846 };
    
```

847

848 The second parameter of the method has the default value MemberStatusEnum.Professional. The parameter
 849 default values have been introduced to support the so-called method extensions. The idea is as flows

- 850 • A derived class may override a method and add a new input parameter
- 851 • The added parameter is declared with a default value
- 852 • A client written against the base class calls the method without that parameter because it does
- 853 not know about it
- 854 • The class implementation does not error out, but takes the default value of the missing
- 855 parameter and executes the “extended” method implementation.

856 The example does not illustrate method overriding to keep it simple. However the
 857 GetNumberOfProfessionals method can be called with two arguments, or only with one out argument.

858 The same mechanism can be used when upgrading a schema, where clients written against a previous
 859 version can call extended method in a new version.

860 Method parameters are identified by name and not by position. Therefore parameters with default values
 861 can be added to the method signature at any position, and clients invoking the method can pass
 862 arguments in any order.

863 Declarations of a CIM class must conform to the following model integrity constraints

Comment [W.K.15]: Not the final list

- 864 • Only qualifiers with the scope including “class” shall be applied to class declarations
- 865 • Classes shall inherit from classes or structures
- 866 • Method names in a class shall be unique
- 867 • Property names in a class shall be unique
- 868 • An abstract class shall not inherit from a concrete class
- 869 • Concrete class shall have a key property
- 870 • There can be only one key property in a class
- 871 • Only qualifiers with the scope including “method” shall be applied to method declarations
- 872 • Only qualifiers with the scope including “parameter” shall be applied to method parameters
- 873 • Method parameters with default value
 - 874 • must form a consecutive trailing group
 - 875 • must have the `in` modifier keyword
- 876 • ...

877 **6.2.4 Association declaration**

878 An association declaration in MOF corresponds to the Association meta-model element defined in
 879 DSP0004 and shall conform to the following associationDeclaration ABNF rule:

```

880 associationDeclaration = [ qualifierList ] *2classModifier
881                        ASSOCIATION associationName [ superAssociation ]
882                        "{" * classFeature "}" ";"
883 associationName       = schemaQualified_name
884 superAssociation     = ":" schemaQualified_name
885 ASSOCIATION          = "association" ; keyword: case insensitive
886
    
```

887 In the meta-model the Association derives from Class. The Association is structurally identical to Class, but
888 its declaration

- 889 • must have a least two, scalar value, reference properties (inherited and defined in the
890 association), and
- 891 • the set of reference properties of the association (inherited and defined in the association)
892 represent the ends of the association.

893

894 The GOLF_MemberLocker below is an example of an association with two ends and it represents an
895 assignment of lockers to club members.

896

```
897 // =====
898 // GOLF_MemberLocker
899 // =====
900 association GOLF_MemberLocker : GOLF_Base {
901     [Max(1)]
902     GOLF_Member REF Member;
903     [Max(1)]
904     GOLF_Locker REF Locker;
905     GOLF_Date AssignedOnDate;
906 };
```

907 Notice that the GOLF_MemberLocker association derives from GOLF_Base, which defines the key property
908 InstanceID. This is a change to the convention established for CIM v2 schemas. In CIM v2 schemas, the
909 key of an association instance was usually the combined value of its reference properties. In CIM v3
910 schemas, it is recommended that association instances are identified the same way as instances of
911 ordinary classes using a single, client-opaque key property.

912

913 Also notice that the multiplicity of the association's ends can be defined using the Max and Min qualifiers
914 (see the discussion of associations in Section 6.4 of DSP0004).

915

916 A declaration of a CIM association must meet the following integrity constraints:

- 917 • Only qualifiers with the scope including "association" shall be applied to association declarations
- 918 • An association can inherit from an association, a structure, and form a class with no reference properties
- 919 • Other class constrains apply to associations
- 920 • An association must have at least two reference properties
- 921 • ...

922

923 6.2.5 CIM Primitive types declarations

924 CIM has the following set of primitive data types. Each MOF primitive data type corresponds to a meta-
925 model element derived from the PrimitiveType element defined in DSP0004.

- 926 • Numeric primitive types
 - 927 • Integers
 - 928 • Reals

Comment [W.K.16]: This is a sensible engineering constraint, but we need to verify if we should allow turning classes to associations.

- 929 • string
- 930 • datetime
- 931 • boolean, and
- 932 • octetstring.

933 A MOF base data type shall conform to the primitiveType ABNF rule:

```

934 primitiveType      = DT_Integer /
935                    DT_REAL32 /
936                    DT_REAL64 /
937                    DT_STRING /
938                    DT_DATETIME /
939                    DT_BOOLEAN /
940                    DT_OCTETSTRING
941 DT_Integer         = DT_UnsignedInteger /
942                    DT_SignedInteger
943 DT_UnsignedInteger = DT_UINT8 /
944                    DT_UINT16 /
945                    DT_UINT32 /
946                    DT_UINT64
947                    DT_UINT128
948 DT_SignedInteger  = DT_SINT8 /
949                    DT_SINT16 /
950                    DT_SINT32 /
951                    DT_SINT64 /
952                    DT_SINT128 /
953 DT_UINT8          = "uint8"           ; keyword: case insensitive
954 DT_UINT16         = "uint16"          ; keyword: case insensitive
955 DT_UINT32         = "uint32"          ; keyword: case insensitive
956 DT_UINT64         = "uint64"          ; keyword: case insensitive
957 DT_UINT128        = "uint128"         ; keyword: case insensitive
958 DT_SINT8          = "sint8"           ; keyword: case insensitive
959 DT_SINT16         = "sint16"          ; keyword: case insensitive
960 DT_SINT32         = "sint32"          ; keyword: case insensitive
961 DT_SINT64         = "sint64"          ; keyword: case insensitive
962 DT_SINT128        = "sint128"         ; keyword: case insensitive

```

```

963 DT_REAL32           = "real32"           ; keyword: case insensitive
964 DT_REAL64           = "real64"           ; keyword: case insensitive
965 DT_DATETIME         = "datetime"         ; keyword: case insensitive
966 DT_STRING           = "string"           ; keyword: case insensitive
967 DT_BOOLEAN          = "boolean"          ; keyword: case insensitive
968 DT_OCTETSTRING      = "octetstring"      ; keyword: case insensitive
    
```

969
 970 The primitive types are used in the declaration of

- 971 • Qualifiers
- 972 • Properties
- 973 • Enumerations
- 974 • Method parameters, and
- 975 • Method results.

976

977 **6.3 Instance and value definitions**

978 The CIM meta-model, and therefore MOF, allow for defining instances of classes and associations, and
 979 values of structures. The instanceDeclaration in MOF maps to the InstanceDeclaration in the meta-model,
 980 and its ABNF rules are as follows:

```

981 instanceDeclaration = INSTANCE OF ( structureName / className / associationName ) [ alias ]
982                   = "{" *propertyValueList ";"
983 propertyValueList  = "{" *propertyValue "}"
984 propertyValue      = propertyName "=" value ";"
985 propertyName      = IDENTIFIER
986 value              = singleValue /
987                   valueArray /
988 singleValue        = simpleValue /
989                   complexValue
990 valueArray         = "{" [ valueArrayElementList ] "}"
991 valueArrayElementList = singleValue *( "," singleValue )
992 simpleValue        = literalPrimitiveTypeValue /
993                   literalObjectPathValue /
994                   aliasIdentifier
995 complexValue       = propertyValueList /
996                   instanceDeclaration
    
```

Comment [W.K.17]: Should we introduce rules to define VALUE OF structureName [ALIAS] ...?

Comment [W.K.18]: This is called Instance in the meta-model

Comment [W.K.19]: The value could be NULL

Comment [W.K.20]: This is questioned; should we allow "instance of X { ..." to the right of the property name

```

997 alias                = AS aliasIdentifier
998 INSTANCE             = "instance"           ; keyword: case insensitive
999 AS                   = "as"                 ; keyword: case insensitive
1000 OF                   = "of"                 ; keyword: case insensitive

```

1001

1002 The example below is a definition of an instance of GOLF_ClubMember, which represents a person with the
1003 name John Doe.

1004 Values of structures can be defined in two ways

- 1005 • By inlining them inside the owner class or structure instance, like the value of the
1006 LastPaymentDate property in the example below, or
- 1007 • By defining them separately and giving them an alias, like the JohnDoesPhonNo and
1008 JohnDoesStartDate values that are first predefined and then used in the definition of the John
1009 Doe instance.

1010

```

1011 // =====
1012 // Instance of GOLF_ClubMember John Doe
1013 // =====
1014
1015 instance of GOLF_Date as JohnDoesStartDate
1016 {
1017     Year = 2011;
1018     Month = July;
1019     Day = 17;
1020 };
1021
1022 instance of GOLF_PhoneNumber as JohnDoesPhoneNo
1023 {
1024     AreaCode = {"9", "0", "7"};
1025     Number = {"7", "4", "7", "4", "8", "8", "4"};
1026 };
1027
1028 instance of GOLF_ClubMember
1029 {
1030     Caption = "Instance of John Doe\'s GOLF_ClubMember object";
1031     FirstName = "John";
1032     LastName = "Doe";
1033     Status = Basic;
1034     MembershipEstablishedDate = JohnDoesStartDate;
1035     MonthlyFee = 250.00;
1036     LastPaymentDate = instance of GOLF_Date
1037     {
1038         Year = 2011;
1039         Month = July;
1040         Day = 31;
1041     };

```

```
1042 MemberAddress = instance of GOLF_Address
1043 {
1044     State = IL;
1045     City = "Oak Park";
1046     Street "Oak Park Av.";
1047     StreetNo = "1177";
1048     AppartmentNo = "3B";
1049 };
1050 MemberPhoneNo = JohnDoesPhoneNo;
1051 MemberEmailAddress = "JonDoe@hotmail.com";
1052 };
```

1053

1054 The literal value, which appears on the right-hand side of the property value definitions, is defined by the
1055 following ABNF rule.

```
1056 literalValue = integerValue / realValue / stringValue / datetimeValue / booleanValue /
1057               enumValue / nullValue / octetstringValue
```

1058

1059 The values of specific primitive types are in [Annex A, A.16](#).

Annex A (normative)
MOF grammar description

1060

1061

1062

1063 The grammar is defined using the ABNF notation described in: <http://tools.ietf.org/html/rfc5234>.

1064

1065 The definition uses the following conventions

1066 - All non-terminals are spelled in lower-case

1067 - Punctuation terminals like ";" are show verbatim

1068 - Terminal symbols are spelled in CAPITL letter when used and then defined in the keywords and symbols
1069 section (they correspond to the lexical tokens)

1070

1071 The grammar is written to be lexically permissive. This means that some of the MOF constraints are
1072 expected to be checked over an in-memory MOF representation specification (the ASTs) after the MOF
1073 file(s) have been parsed. For example the constraint that enumeration elements in a derived enumeration
1074 must be of the same type as the base enumeration is not encoded in the grammar. Similarly the default
1075 values of qualifier definitions are lexically permissive to keep the parsing simple.

1076 The grammar is also written with the assumption that MOF v2 files will be converted to MOF v3 before
1077 they can be parsed by the MOF v3 compiler. For example we assume that all ValueMap and Values
1078 qualifier pairs will be translated to enumerations before a parser implementing the following grammar can
1079 be used.

1080

1081 **A.1 MOF specification**

```

1082 mofCompilationUnit      = *mofProduction
1083 mofProduction           = compilerDirective /
1084                          structureDeclaration /
1085                          classDeclaration /
1086                          associationDeclaration /
1087                          enumerationDeclaration /
1088                          instanceDeclaration /
1089                          qualifierDeclaration

```

1090

1091 **A.2 Compiler directive**

```

1092 compilerDirective       = PRAGMA (pragmaName / standardPragmaName )
1093                          "(" pragmaParameter ")"
1094 pragmaName              = IDENTIFIER
1095 standardPragmaName     = INCLUDE
1096 pragmaParameter        = stringValue
1097 PRAGMA                  = "#pragma"           ; keyword: case insensitive
1098 INCLUDE                 = "include"           ; keyword: case insensitive

```

1099

1100 **A.3 Structure declaration**

1101 The syntactic difference between global and nested structure declarations is that the global declarations must use
 1102 schema-qualified names. This constraint can be verified after the MOF files have been parsed into the corresponding
 1103 abstract syntax trees.

```

1104 structureDeclaration    = [ qualifierList ] [FINAL]
1105                          STRUCTURE structureName [superStructure]
1106                          "{" *propertyDeclaration "}" ";"
1107 structureName           = ( IDENTIFIER / schemaQualifiedName )
1108 superStructure         = ":" ( IDENTIFIER / schemaQualifiedName )
1109 FINAL                   = "final"             ; keyword: case insensitive
1110 STRUCTURE               = "structure"         ; keyword: case insensitive

```

1111

1112 **A.4 Class declaration**

```

1113 classDeclaration      = [ qualifierList ] *2classModifier
1114                       [ABSTRACT] CLASS className [ superClass ]
1115                       "{" * classFeature "}" ";"
1116 classModifier         = ABSTRACT / FINAL
1117 className             = schemaQualified_name
1118 superClass            = ":" schemaQualified_name
1119 classFeature          = propertyDeclaration /
1120                       enumerationDeclaration /
1121                       structureDeclaration /
1122                       methodDeclaration
1123 ABSTRACT              = "abstract"           ; keyword: case insensitive
1124 CLASS                 = "class"             ; keyword: case insensitive
1125

```

1126 **A.5 Association declaration**

1127 The only syntactic difference between the class and the association is the use of the keyword "association".

```

1128 associationDeclaration = [ qualifierList ] *2classModifier
1129                       ASSOCIATION associationName [ superAssociation ]
1130                       "{" * classFeature "}" ";"
1131 associationName        = schemaQualified_name
1132 superAssociation       = ":" schemaQualified_name
1133 ASSOCIATION           = "association"        ; keyword: case insensitive
1134

```

1135 **A.6 Enumeration declaration**

1136 The grammar does not differentiate between derived integer and string enumerations. This is because syntactically
1137 they will be the same if literals are given no values.

```

1138 enumDeclaration       = [ qualifierList ] [FINAL]
1139                       ENUMERATION enumName ":"
1140                       ( integerEnumDeclaration /
1141                       stringEnumDeclaration /
1142                       derivedEnumDeclaration )
1143 enumName              = ( IDENTIFIER / schemaQualified_name )
1144 integerEnumDeclaration = DT_UnsignedInteger "{" integerEnumElement *( "," integerEnumElement )" ";"
1145 stringEnumDeclaration = DT_STRING "{" stringEnumElement *( "," stringEnumElement )" ";"

```

Comment [W.K.21]: Should we change the separator to " " ?

1146	integerEnumElement	= [qualifierList] enumLiteral "=" integerValue
1147	stringEnumElement	= [qualifierList] enumLiteral ["=" stringValue]
1148	derivedEnumDeclaration	= (IDENTIFIER / schemaQualifiedName)
1149		"{" enumElement *("," enumElement)"}" ";"
1150	enumElement	= enumLiteral ["=" (integerValue / stringValue)]
1151	enumLiteral	= IDENTIFIER
1152	ENUMERATION	= "enumeration"
1153		

Comment [W.K.22]: Modeled the values of integer enumerations as non-optional.

Comment [W.K.23]: Left string values optional.

1154 **A.7 Instance declaration**

1155 The grammar is not attempting to verify that the type of the value that is part of a property declaration is consistent
1156 with the type of the property.

1157 Notice that by the definition of the *valueArray* an array cannot contains another array; it can only contain a collection
1158 of single-value elements.

1159	instanceDeclaration	= INSTANCE OF (structureName / className / associationName) [alias]
1160		propertyValueList ";"
1161	propertyValueList	= "{" *propertyValue "}"
1162	propertyValue	= propertyName "=" value ";"
1163	propertyName	= IDENTIFIER
1164	value	= singleValue /
1165		valueArray /
1166	singleValue	= simpleValue /
1167		complexValue
1168	valueArray	= "{" [valueArrayElementList] "}"
1169	valueArrayElementList	= singleValue *("," singleValue)
1170	simpleValue	= literalPrimitiveTypeValue /
1171		literalObjectPathValue /
1172		aliasIdentifier
1173	complexValue	= propertyValueList /
1174		instanceDeclaration
1175	alias	= AS aliasIdentifier
1176	INSTANCE	= "instance" ; keyword: case insensitive
1177	AS	= "as" ; keyword: case insensitive
1178	OF	= "of" ; keyword: case insensitive
1179		

Comment [W.K.24]: This is called Instance in the meta-model

Comment [W.K.25]: The value could be NULL

Comment [W.K.26]: This is questioned; should we allow "instance of X { ..." to the right of the property name

1180 The type of the value associated with a property should be consistent with the type of the property. For example the
1181 default value of a structure containing string and integer properties should be a string and an integer.

1182 **A.8 Qualifier declaration**

Comment [W.K.27]: Align the syntax of the qualifier declaration with class format

1183 Notice that qualifiers can be qualified themselves. This is to allow run-time analysis of such qualifier properties as
 1184 their propagation rules.

1185 Since many v2 qualifiers have been replaced with keywords, the MOF v2 files will have to be converted to MOF v3
 1186 before parsing.

```

1187 qualifierDeclaration      = [ qualifierList ] [TRANSLATABLE]
1188                           QUALIFIER qualifierName ":" qualifierType qualifierScope [qualifierPolicy] ":"
1189 qualifierName             = IDENTIFIER
1190 qualifierType             = singleValueQualifierType / arrayValueQualifierType
1191 singleValueQualifierType = ( primitiveType / enumName ) [ "=" defaultQualifierValue ]
1192 arrayValueQualifierType  = primitiveType "[" "]" [ "=" defaultQualifierValueList ]
1193 defaultQualifierValue    = literalValue
1194 defaultQualifierValueList = "{" primitiveTypeValueList "}"
1195 primitiveTypeValueList   = literalPrimitiveTypeValue *( "," literalValue)
1196 qualifierScope           = "," SCOPE "(" ANY / qualifiedElementList ")"
1197 qualifierPolicy          = "," POLICY "(" APPLYMANY / APPLYONCE / RESTRICTED ")"
1198 qualifiedElementList     = qualifiedElement *( "," qualifiedElement )
1199 qualifiedElement         = STRUCTURE / CLASS / ASSOCIATION /
1200                           ENUMERATION / ENUMERATIONLITERAL /
1201                           PROPERTY / REFPROPERTY /
1202                           METHOD / PARAMETER /
1203                           QUALIFIER
1204 TRANSLATABLE            = "translatable"           ; keyword: case insensitive
1205 SCOPE                   = "scope"                 ; keyword: case insensitive
1206 ANY                     = "any"                   ; keyword: case insensitive
1207 POLICY                  = "policy"                ; keyword: case insensitive
1208 APPLYMANY               = "applymany"            ; keyword: case insensitive
1209 APPLYONCE               = "applyonce"            ; keyword: case insensitive
1210 RESTRICTED              = "restricted"           ; keyword: case insensitive
1211 ENUMERATIONLITERAL     = "enumerationliteral"    ; keyword: case insensitive
1212 PROPERTY                = "property"             ; keyword: case insensitive
1213 REFPROPERTY            = "reference"             ; keyword: case insensitive
1214 METHOD                  = "method"                ; keyword: case insensitive
1215 PARAMETER              = "parameter"            ; keyword: case insensitive
1216 QUALIFIER               = "qualifier"           ; keyword: case insensitive
    
```

Comment [W.K.28]: What is the default value if the policy is not specified? ApplyMany?

1217

1218 **A.9 Qualifier list**

1219	qualifierList	=	"[" qualifier *("," qualifier) "]"
1220	qualifier	=	qualifierName [qualifierParameter]
1221	qualifierName	=	IDENTIFIER
1222	qualifierParameter	=	qualifierValueInitializer / qualifierValueArrayInitializer
1223	qualifierValueInitializer	=	"(" literalPrimitiveTypeValue ")"
1224	qualifierValueArrayInitializer	=	"{" primitiveTypeValue *("," primitiveTypeValue)" }

1225

1226 **A.10 Property declaration**

1227	propertyDeclaration	=	[qualifierList] [KEY]
1228			(basePropertyDeclaration /
1229			structurePropertyDeclaration /
1230			enumPropertyDeclaration /
1231			classPropertyDeclaration /
1232			referencePropertyDeclaration)
1233	basePropertyDeclaration	=	primitiveType propertyName [array] ["=" primitiveDefaultValue] ";"
1234	structurePropertyDeclaration	=	structureName propertyName [array] ["=" structureDefaultValue] ";"
1235	enumPropertyDeclaration	=	enumName propertyName [array] ["=" enumDefaultValue] ";"
1236	classPropertyDeclaration	=	className propertyName [array] ["=" structureDefaultValue] ";"
1237	referencePropertyDeclaration	=	className REF propertyName [array] ["=" referenceDefaultValue] ";"
1238	array	=	"[" "]"
1239	propertyName	=	IDENTIFIER
1240	KEY	=	"key" ; keyword: case insensitive
1241	REF	=	"ref" ; keyword: case insensitive

Comment [W.K.29]: Array size constraints should be specified using the Constraint qualifier => OCL.

1242

1243 **A.11 Method declaration**

1244	methodDeclaration	=	[qualifierList] [STATIC] (dataType [array] / VOID) methodName
1245			"(" [parameterList] ")" ";"
1246	methodName	=	IDENTIFIER
1247	VOID	=	"void" ; keyword: case insensitive
1248	STATIC	=	"static" ; keyword: case insensitive
1249	parameterList	=	parameterDeclaration *("," parameterDeclaration)

1250

1251 **A.12 Parameter declaration**

1252	parameterDeclaration	=	[qualifierList] [IN / OUT / INOUT]
1253			(primitiveParamDeclaration /
1254			structureParamDeclaration /
1255			enumParamDeclaration /
1256			classParamDeclaration /
1257			referenceParamDeclaration)
1258	primitiveParamDeclaration	=	primitiveType parameterName [array] ["=" primitiveDefaultValue]
1259	structureParamDeclaration	=	structureName parameterName [array] ["=" structureDefaultValue]
1260	enumParamDeclaration	=	enumName parameterName [array] ["=" enumDefaultValue]
1261	classParamDeclaration	=	className parameterName [array] ["="structureDefaultValue]
1262	referenceParamDeclaration	=	className "REF" parameterName [array] ["=" referenceDefaultValue]
1263	parameterName	=	IDENTIFIER
1264	IN	=	"in" ; keyword: case insensitive
1265	OUT	=	"out" ; keyword: case insensitive
1266	INOUT	=	"inout" ; keyword: case insensitive

1267

1268 **A.13 Values**

1269	primitiveDefaultValue	=	literalPrimitiveTypeValue / literalValueArray
1270	literalValueArray	=	"{" literalPrimitiveTypeValue *("," literalPrimitiveTypeValue)}"
1271	literalValue	=	integerValue / realValue / stringValue / datetimeValue / booleanValue /
1272			enumValue / nullValue / octetstringValue
1273	integerValue	=	binaryValue / octalSignedValue / decimalValue / hexValue
1274	octetstringValue	=	octalValue
1275	enumValue	=	[enumName"."] enumLiteral
1276	structureDefaultValue	=	complexValue / complexValueArray
1277	complexValueArray	=	"{" complexValue *("," complexValue)}"
1278	enumDefaultValue	=	stringValue
1279	referenceDefaultValue	=	referenceValue / referenceValueArray
1280	referenceValue	=	literalObjectPathValue / aliasIdentifier
1281	referenceValueArray	=	"{" referenceValue *("," referenceValue)}"
1282	literalObjectPathValue	=	stringValue

Comment [W.K.30]: Should we define the structure of the object path string in MOF?

1283

1284 **A.14 Base data types**

1285	primitiveType	=	DT_Integer /	
1286			DT_REAL32 /	
1287			DT_REAL64 /	
1288			DT_STRING /	
1289			DT_DATETIME /	
1290			DT_BOOLEAN /	
1291			DT_OCTETSTRING	
1292	DT_Integer	=	DT_UnsignedInteger /	
1293			DT_SignedInteger	
1294	DT_UnsignedInteger	=	DT_UINT8 /	
1295			DT_UINT16 /	
1296			DT_UINT32 /	
1297			DT_UINT64	
1298			DT_UINT128	
1299	DT_SignedInteger	=	DT_SINT8 /	
1300			DT_SINT16 /	
1301			DT_SINT32 /	
1302			DT_SINT64 /	
1303			DT_SINT128 /	
1304	DT_UINT8	=	"uint8"	; keyword: case insensitive
1305	DT_UINT16	=	"uint16"	; keyword: case insensitive
1306	DT_UINT32	=	"uint32"	; keyword: case insensitive
1307	DT_UINT64	=	"uint64"	; keyword: case insensitive
1308	DT_UINT128	=	"uint128"	; keyword: case insensitive
1309	DT_SINT8	=	"sint8"	; keyword: case insensitive
1310	DT_SINT16	=	"sint16"	; keyword: case insensitive
1311	DT_SINT32	=	"sint32"	; keyword: case insensitive
1312	DT_SINT64	=	"sint64"	; keyword: case insensitive
1313	DT_SINT128	=	"sint128"	; keyword: case insensitive
1314	DT_REAL32	=	"real32"	; keyword: case insensitive
1315	DT_REAL64	=	"real64"	; keyword: case insensitive

1316 DT_DATETIME = "datetime" ; keyword: case insensitive
 1317 DT_STRING = "string" ; keyword: case insensitive
 1318 DT_BOOLEAN = "boolean" ; keyword: case insensitive
 1319 DT_OCTETSTRING = "octetstring" ; keyword: case insensitive

1320

1321 A.15 Names

1322 The values recognized by the following are case independent. They also don't allow whitespace.

1323 schemaQualifiedName = schemaName UNDERSCORE IDENTIFIER
 1324 schemaName = firstSchemaChar *(nextSchemaChar)
 1325 firstSchemaChar = UPPERALPHA / LOWERALPHA
 1326 nextSchemaChar = firstSchemaChar / decimalDigit
 1327
 1328 IDENTIFIER = firstIdentifierChar *(nextIdentifierChar)
 1329 firstIdentifierChar = UPPERALPHA / LOWERALPHA / UNDERSCORE
 1330 nextIdentifierChar = firstIdentifierChar / decimalDigit
 1331 aliasIdentifier = "\$" IDENTIFIER

1332

1333 A.16 Primitive type values

1334 ; Whitespace is allowed between the double-quoted parts.
 1335 ; The combined date-time string value shall conform to the format defined by the dt-format ABNF rule,
 1336 ; with further constraints on the field values as defined in DSP0004.
 1337 datetimeValue = DOUBLEQUOTE dt-format DOUBLEQUOTE
 1338 dt-format = dt-timestampValue / dt-intervalValue
 1339
 1340 dt-timestampValue = 14*14(decimalDigit) "." dt-microseconds ("+" / "-") dt-timezone
 1341 / dt-yyyyymmddhhmmss "." 6*6("*") ("+" / "-") dt-timezone
 1342 dt-yyyyymmddhhmmss = 12*12(decimalDigit) 2*2("*")
 1343 / 10*10(decimalDigit) 4*4("*")
 1344 / 8*8(decimalDigit) 6*6("*")
 1345 / 6*6(decimalDigit) 8*8("*")
 1346 / 4*4(decimalDigit) 10*10("*")
 1347 / 14*14("*")

1348	dt-timezone	=	3*3(decimalDigit)	
1349				
1350	dt-intervalValue	=	14*14(decimalDigit) "." dt-microseconds ":" "000" /	
1351			dt-dddddddhhmmss "." 6*6("**") ":" "000"	
1352	dt-dddddddhhmmss	=	12*12(decimalDigit) 2*2("**")	
1353			/ 10*10(decimalDigit) 4*4("**")	
1354			/ 8*8(decimalDigit) 6*6("**")	
1355			/ 14*14("**")	
1356	dt-microseconds	=	6*6(decimalDigit)	
1357			/ 5*5(decimalDigit) 1*1("**")	
1358			/ 4*4(decimalDigit) 2*2("**")	
1359			/ 3*3(decimalDigit) 3*3("**")	
1360			/ 2*2(decimalDigit) 4*4("**")	
1361			/ 1*1(decimalDigit) 5*5("**")	
1362			/ 6*6("**")	
1363				
1364	; Whitespace and comment is allowed between double quoted parts.			
1365	; Double quotes shall be escaped. Tthe (unescaped) contents of stringValue shall conform to the			
1366	; string representation for object paths.			
1367	stringValue	=	1*(DOUBLEQUOTE *stringChar DOUBLEQUOTE)	
1368	stringChar	=	UCScharString / stringEscapeSequence	
1369	UCScharString	=	1*6(hexDigit) ; any UCS character for use in string constants.	
1370	stringEscapeSequence	=	BACKSLASH (BACKSPACE / TAB / LINEFEED / FORMFEED / RETURN /	
1371			DOUBLEQUOTE / SINGLEQUOTE / BACKSLASH / UCSchar)	
1372	BACKSPACE	=	%x08 ; U+0008: backspace	
1373	TAB	=	%x09 ; U+0009: horizontal tab	
1374	LINEFEED	=	%x0A ; U+000A: linefeed	
1375	FORMFEED	=	%x0C ; U+000C: form feed	
1376	RETURN	=	%x0D ; U+000D: carriage return	
1377	DOUBLEQUOTE	=	%x22 ; U+0022: double quote (")	
1378	SINGLEQUOTE	=	%x27 ; U+0027: single quote (')	
1379	BACKSLASH	=	%x5C ; U+005C: backslash (\)	
1380	UPPERALPHA	=	%x41-5A ; U+0041-U+005A or "A" ... "Z"	
1381	LOWERALPHA	=	%x61-7A ; U+0061-U+007A or "a" ... "z"	

1382 UNDERSCORE = %x5F ; U+005F or "_"

1383 UCSchar = ("x" / "X") 1*4(hexDigit) ; no whitespace: UCS code position

1384

1385 ; The following ABNF rules do not allow whitespace, unless stated otherwise.

1386 booleanValue = TRUE / FALSE

1387 FALSE = "false" ; keyword: case Insensitive

1388 TRUE = "true" ; keyword: case Insensitive

1389

1390 ; The following ABNF rules do not allow whitespace, unless stated otherwise:

1391 nullValue = NULL

1392 NULL = "null" ; keyword: case Insensitive

1393

1394 ; The following ABNF rules do not allow whitespace, unless stated otherwise.

1395 binaryValue = ["+" / "-"] 1*binaryDigit ("b" / "B")

1396 binaryDigit = "0" / "1"

1397 octalValue = "0" 1*octalDigit

1398 octalSignedValue = ["+" / "-"] octalValue

1399 octalDigit = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7"

1400 hexValue = ["+" / "-"] ("0x" / "0X") 1*hexDigit

1401 hexDigit = decimalDigit / "a" / "A" / "b" / "B" / "c" / "C" /

1402 "d" / "D" / "e" / "E" / "f" / "F"

1403 decimalValue = ["+" / "-"] (positiveDecimalDigit *decimalDigit / "0")

1404 realValue = ["+" / "-"] *decimalDigit "." 1*decimalDigit

1405 [("e" / "E") ["+" / "-"] 1*decimalDigit]

1406 decimalDigit = "0" / positiveDecimalDigit

1407 positiveDecimalDigit = "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"

1408

Annex B (normative) MOF keywords

1409 Below are the MOF v3 keyword listed in alphabetical order.

1410

#pragma	true
abstract	scope
any	sint8
applymany	sint16
applyonce	sint32
as	sint64
association	sint128
boolean	static
class	string
datetime	structure
enumeration	translatable
enumerationliteral	uint8
false	uint16
in	uint32
include	uint64
inout	uint128
key	void
method	qualifier
null	
octetstring	
of	
out	
parameter	
policy	
property	
real32	
real64	
restricted	
ref	
reference	

1411

Annex C (informative) Example MOF specification

1412
1413
1414

1415 The following is the content of the MOF files in the example GOLF model specification.

1416 C.1 GOLF_Schema.mof

```

1417 //
1418 // Copyright 2011 Distributed Management Task Force, Inc. (DMTF).
1419 // Exampel domain used to illustrate CIM v3 and MOF v3 features
1420 //
1421 #pragma include ("GOLF_Base.mof")
1422 #pragma include ("GOLF_Club.mof")
1423 #pragma include ("GOLF_ClubMember.mof")
1424 #pragma include ("GOLF_Professional.mof")
1425 #pragma include ("GOLF_Locker.mof")
1426 #pragma include ("GOLF_MemberLocker.mof")
1427 #pragma include ("GOLF_Lesson.mof")
1428 //
1429 // Global structures
1430 //
1431 #pragma include ("GlobalStructs/GOLF_Address.mof")
1432 #pragma include ("GlobalStructs/GOLF_Date.mof")
1433 #pragma include ("GlobalStructs/GOLF_PhoneNumber.mof")
1434 //
1435 // Global enumerations
1436 //
1437 #pragma include ("GlobalEnums/GOLF_ResultCodeEnum.mof")
1438 #pragma include ("GlobalEnums/GOLF_GOLF_MonthsEnum.mof")
1439 #pragma include ("GlobalEnums/GOLF_GOLF_StatesEnum.mof")
1440 //
1441 // Instances
1442 //
1443 #pragma include ("Instances/JohnDoe.mof")

```

1444 C.2 GOLF_Base.mof

```

1445 // =====
1446 // GOLF_Base
1447 // =====
1448 abstract class GOLF_Base {
1449     [Description (
1450         "InstanceID is a property that opaquely and uniquely identifies "
1451         "an instance of a class that derives from the GOLF_Base class. ") ]
1452     key string InstanceID;
1453
1454     [Description (

```

```

1455     "A a short textual description (one- line string) of the "
1456     "instance." ),
1457     MaxLen (64)]
1458     string Caption;
1459 };

```

1460 C.3 GOLF_Club.mof

```

1461 // =====
1462 // GOLF_Club
1463 // =====
1464 [Description (
1465     "Instances of this class represent golf clubs. A golf club is an "
1466     "an organization that provides member services to golf players "
1467     "bouth amateur and professional." )]
1468 class GOLF_Club: GOLF_Base {
1469 // ===== properties =====
1470     string ClubName;
1471     GOLF_Date YearEstablished;
1472
1473     GOLF_Address ClubAddress;
1474     GOLF_Phone ClubPhoneNo;
1475     GOLF_Fax ClubFaxNo;
1476     string ClubWebSiteURL;
1477
1478     GOLF_ClubMember REF AllMembers[];
1479     GOLF_Professional REF Professionals[];
1480
1481 // ===== methods =====
1482     static GOLF_ResultSetEnum AddNonProfessionalMember (
1483         in GOLF_ClubMember newMember
1484     );
1485
1486     static GOLF_ResultSetEnum AddProfessional (
1487         in GOLF_Professional newProfessional
1488     );
1489
1490     static UInt32 GetMembersWithOutstandingFees (
1491         in GOLF_Date referenceDate,
1492         out GOLF_ClubMember REF lateMembers[]
1493     );
1494
1495     static GOLF_ResultSetEnum TerminateMembership (
1496         in GOLF_ClubMember REF member
1497     );
1498 };

```

1499 C.4 GOLF_ClubMember.mof

```

1500 // =====

```

```

1501 // GOLF_ClubMember
1502 // =====
1503 [Description (
1504     "Instances of this class represent members of a golf club." )]
1505 class GOLF_ClubMember: GOLF_Base {
1506
1507 // ===== embeded enumerations =====
1508     enumeration MemberStatusEnum : Uint16 {
1509         Basic = 0,
1510         Extended = 1,
1511         VP = 2
1512     };
1513
1514 // ===== properties =====
1515     string FirstName;
1516     string LastName;
1517     MemberStatusEnum Status;
1518     GOLF_Date MembershipEstablishedDate;
1519
1520     Real32 MembershipSignUpFee;
1521     Real32 MonthlyFee;
1522     GOLF_Date LastPaymentDate;
1523
1524     GOLF_Address MemberAddress;
1525     [ Constraint ("inv: not oclIsUndefined()") ]
1526     GOLF_PhoneNumber MemberPhoneNo;
1527     string MemberEmailAddress;
1528
1529 // ===== methods =====
1530     GOLF_ResultCodeEnum SendPaymentReminderMessage();
1531 };

```

1532 C.5 GOLF_Professional.mof

```

1533 // =====
1534 // GOLF_Professional
1535 // =====
1536 class GOLF_Professional : GOLF_ClubMember {
1537 // ===== embeded structures =====
1538     structure Sponsor {
1539         string Name,
1540         GOLF_Date ContractSignedDate;
1541         Real32 ContractAmount;
1542     };
1543
1544 // ===== embeded enumerations =====
1545     enumeration ProfessionalStatusEnum : MemberStatusEnum {
1546         Professional = 6,
1547         SponsoredProfessional = 7

```

```

1548     };
1549
1550 // ===== properties =====
1551     [Override]
1552     ProfessionalStatusEnum Status = Professional;
1553     Sponsor Sponsors[];
1554     Boolean Ranked;
1555
1556 // ===== methods =====
1557     static GOLF_ResultCodeEnum GetNumberOfProfessionals (
1558         out UInt32 NoOfPros,
1559         in ProfessionalStatusEnum = Professional
1560     )
1561 };
1562

```

1563 C.6 GOLF_Locker.mof

```

1564 // =====
1565 // GOLF_Locker
1566 // =====
1567 class GOLF_Locker : GOLF_Base {
1568     string Location;
1569     UInt16 LockerNo;
1570     Real32 MonthlyRentFee;
1571 };

```

1572 C.7 GOLF_MemberLocker.mof

```

1573 // =====
1574 // GOLF_MemberLocker
1575 // =====
1576 association GOLF_MemberLocker : GOLF_Base {
1577     [Max(1)]
1578     GOLF_Member REF Member;
1579     [Max(1)]
1580     GOLF_Locker REF Locker;
1581     GOLF_Date AssignedOnDate;
1582 };

```

1583 C.8 GOLF_Lesson.mof

```

1584 // =====
1585 // GOLF_Lesson
1586 // =====
1587 association GOLF_Lesson : GOLF_Base {
1588     GOLF_Professional REF Instructur;
1589     GOLF_ClubMember REF Student;
1590
1591     DateTime Schedule;
1592     [Description ( "The duration of the lesson in minutes" )]

```

```

1593     UInt16 Duration;
1594     String Location;
1595     Real32 LessonFee;
1596 };

```

1597 C.9 GOLF_Address.mof

```

1598 // =====
1599 // GOLF_Address
1600 // =====
1601 structure GOLF_Address {
1602     GOLF_StateEnum State;
1603     string City;
1604     string Street;
1605     string StreetNo;
1606     string ApartmentNo;
1607 };

```

1608 C.10 GOLF_Date.mof

```

1609 // =====
1610 // GOLF_Date
1611 // =====
1612 structure GOLF_Date {
1613     UInt16 Year = 2000;
1614     GOLF_MonthsEnum Month = GOLF_MonthsEnum.January;
1615     [MinValue(1), MaxValue(31)]
1616     UInt16 Day = 1;
1617 };

```

1618 C.11 GOLF_PhoneNumber.mof

```

1619 // =====
1620 // GOLF_PhoneNumber
1621 // =====
1622 structure GOLF_PhoneNumber {
1623     [Constraint ("inv: size() = 3")]
1624     Char16 AreaCode[];
1625     [Constraint ("inv: size() = 7")]
1626     Char16 Number[];
1627 };

```

1628 C.12 GOLF_ResultCodeEnum.mof

```

1629 // =====
1630 // GOLF_ResultCodeEnum
1631 // =====
1632 enumeration GOLF_ResultCodeEnum : UInt32 {
1633     // The operation was successful
1634     RESULT_OK = 0,
1635     // A general error occurred, not covered by a more specific error code.
1636     RESULT_FAILED = 1,
1637     // Access to a CIM resource is not available to the client.

```

```
1638 RESULT_ACCESS_DENIED = 2,
1639 // The target namespace does not exist.
1640 RESULT_INVALID_NAMESPACE = 3,
1641 // One or more parameter values passed to the method are not valid.
1642 RESULT_INVALID_PARAMETER = 4,
1643 // The specified class does not exist.
1644 RESULT_INVALID_CLASS = 5,
1645 // The requested object cannot be found.
1646 RESULT_NOT_FOUND = 6,
1647 // The requested operation is not supported.
1648 RESULT_NOT_SUPPORTED = 7,
1649 // The operation cannot be invoked because the class has subclasses.
1650 RESULT_CLASS_HAS_CHILDREN = 8,
1651 // The operation cannot be invoked because the class has instances.
1652 RESULT_CLASS_HAS_INSTANCES = 9,
1653 // The operation cannot be invoked because the superclass does not exist.
1654 RESULT_INVALID_SUPERCLASS = 10,
1655 // The operation cannot be invoked because an object already exists.
1656 RESULT_ALREADY_EXISTS = 11,
1657 // The specified property does not exist.
1658 RESULT_NO_SUCH_PROPERTY = 12,
1659 // The value supplied is not compatible with the type.
1660 RESULT_TYPE_MISMATCH = 13,
1661 // The query language is not recognized or supported.
1662 RESULT_QUERY_LANGUAGE_NOT_SUPPORTED = 14,
1663 // The query is not valid for the specified query language.
1664 RESULT_INVALID_QUERY = 15,
1665 // The extrinsic method cannot be invoked.
1666 RESULT_METHOD_NOT_AVAILABLE = 16,
1667 // The specified extrinsic method does not exist.
1668 RESULT_METHOD_NOT_FOUND = 17,
1669 // The specified namespace is not empty.
1670 RESULT_NAMESPACE_NOT_EMPTY = 20,
1671 // The enumeration identified by the specified context is invalid.
1672 RESULT_INVALID_ENUMERATION_CONTEXT = 21,
1673 // The specified operation timeout is not supported by the CIM Server.
1674 RESULT_INVALID_OPERATION_TIMEOUT = 22,
1675 // The Pull operation has been abandoned.
1676 RESULT_PULL_HAS_BEEN_ABANDONED = 23,
1677 // The attempt to abandon a concurrent Pull operation failed.
1678 RESULT_PULL_CANNOT_BE_ABANDONED = 24,
1679 // Using a filter in the enumeration is not supported by the CIM server.
1680 RESULT_FILTERED_ENUMERATION_NOT_SUPPORTED = 25,
1681 // The CIM server does not support continuation on error.
1682 RESULT_CONTINUATION_ON_ERROR_NOT_SUPPORTED = 26,
1683 // The operation failed because server limits were exceeded.
1684 RESULT_SERVER_LIMITS_EXCEEDED = 27,
1685 // The CIM server is shutting down and cannot process the operation.
```



```
1686 RESULT_SERVER_IS_SHUTTING_DOWN = 28
1687 };
```

1688 C.13 GOLF_MonthsEnum.enum

```
1689 // =====
1690 // GOLF_MonthsEnum
1691 // =====
1692 enumeration GOLF_MonthsEnum : String {
1693     January,
1694     February,
1695     March,
1696     April,
1697     May,
1698     June,
1699     July,
1700     August,
1701     September,
1702     October,
1703     November,
1704     December
1705 };
```

1706 C.14 GOLF_StatesEnum.mof

```
1707 // =====
1708 // GOLF_StatesEnum
1709 // =====
1710 enumeration GOLF_StatesEnum : string {
1711     AL = "Alabama",
1712     AK = "Alaska",
1713     AZ = "Arizona",
1714     AR = "Arkansas",
1715     CA = "California",
1716     CO = "Colorado",
1717     CT = "Connecticut",
1718     DE = "Delaware",
1719     FL = "Florida",
1720     GA = "Georgia",
1721     HI = "Hawaii",
1722     ID = "Idaho",
1723     IL = "Illinois",
1724     IN = "Indiana",
1725     IA = "Iowa",
1726     KS = "Kansas",
1727     LA = "Louisiana",
1728     ME = "Main",
1729     MD = "Maryland",
1730     MA = "Massachusetts",
1731     MI = "Michigan",
```

```

1732     MS = "Mississippi",
1733     MO = "Missouri",
1734     MT = "Montana",
1735     NE = "Nebraska",
1736     NV = "Nevada",
1737     NH = "New Hampshire",
1738     NJ = "New Jersey",
1739     NM = "New Mexico",
1740     NY = "New York",
1741     NC = "North Carolina",
1742     ND = "North Dakota",
1743     OH = "Ohio",
1744     OK = "Oklahoma",
1745     OR = "Oregon",
1746     PA = "Pennsylvania",
1747     RI = "Rhode Island",
1748     SC = "South Carolina",
1749     SD = "South Dakota",
1750     TX = "Texas",
1751     UT = "Utah",
1752     VT = "Vermont",
1753     VA = "Virginia",
1754     WA = "Washington",
1755     WV = "West Virginia",
1756     WI = "Wisconsin",
1757     WY = "Wyoming"
1758 };
    
```

1759 **C.15 JohnDoe.mof**

```

1760 // =====
1761 // Instance of GOLF_ClubMember John Doe
1762 // =====
1763
1764 instance of GOLF_Date as JohnDoesStartDate
1765 {
1766     Year = 2011;
1767     Month = July;
1768     Day = 17;
1769 };
1770
1771 instance of GOLF_PhoneNumber as JohnDoesPhoneNo
1772 {
1773     AreaCode = {"9", "0", "7"};
1774     Number = {"7", "4", "7", "4", "8", "8", "4"};
1775 };
1776
1777 instance of GOLF_ClubMember
1778 {
    
```

```
1779     Caption = "Instance of John Doe\'s GOLF_ClubMember object";
1780     FirstName = "John";
1781     LastName = "Doe";
1782     Status = Basic;
1783     MembershipEstablishedDate = JohnDoesStartDate;
1784     MonthlyFee = 250.00;
1785     LastPaymentDate = instance of GOLF_Date
1786     {
1787         Year = 2011;
1788         Month = July;
1789         Day = 31;
1790     };
1791     MemberAddress = instance of GOLF_Address
1792     {
1793         State = IL;
1794         City = "Oak Park";
1795         Street "Oak Park Av.";
1796         StreetNo = "1177";
1797         AppartmentNo = "3B";
1798     };
1799     MemberPhoneNo = JohnDoesPhoneNo;
1800     MemberEmailAddress = "JonDoe@hotmail.com";
1801 };
1802
```