



Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Committee Draft 01, Revision 43

~~186 December~~ January 2008

Specification URIs:

This Version:

- <http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01-rev42.html>
- <http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01-rev24.doc>
- <http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01-rev42.pdf>

Previous Version:

Latest Version:

- <http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec.html>
- <http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec.doc>
- <http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec.pdf>

Latest Approved Version:

Technical Committee:

OASIS Service Component Architecture / J (SCA-J) TC

Chair(s):

Simon Nash,	IBM
Michael Rowley,	BEA Systems
Mark Combellack,	Avaya

Editor(s):

Ron Barack,	SAP
David Booz,	IBM
Mark Combellack,	Avaya
Mike Edwards,	IBM
Anish Karmarkar,	Oracle
Ashok Malhotra,	Oracle
Peter Peshev,	SAP

Related work:

This specification replaces or supersedes:

- Service Component Architecture Java Annotations and APIs Specification Version 1.00, March 21 2007

This specification is related to:

- Service Component Architecture Assembly Model Specification Version 1.1
- Service Component Architecture Policy Framework Specification Version 1.1

Declared XML Namespace(s):

<http://docs.oasis-open.org/ns/opencsa/sca/200712>

Abstract:

The SCA Java Common Annotation and APIs specify a Java syntax for programming concepts defined in the SCA Assembly Model Specification. It specifies a set of APIs and annotations that may be used by Java-based SCA specifications.

Specifically, this specification covers:

1. Implementation metadata for specifying component services, references, and properties
2. A client and component API
3. Metadata for asynchronous and conversational services
4. Metadata for callbacks
5. Definitions of standard component implementation scopes
6. Java to WSDL and WSDL to Java mappings
7. Security policy annotations

Note that individual programming models may chose to implement their own mappings of assembly model concepts using native APIs and idioms when appropriate.

Status:

This document was last revised or approved by the OASIS Service Component Architecture / J (SCA-J) TC on the above date. The level of approval is also listed above. Check the “Latest Version” or “Latest Approved Version” location noted above for possible later revisions of this document.

Technical Committee members should send comments on this specification to the Technical Committee’s email list. Others should send comments to the Technical Committee by using the “Send A Comment” button on the Technical Committee’s web page at <http://www.oasis-open.org/committees/sca-j/>.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (<http://www.oasis-open.org/committees/sca-j/ipr.php>).

The non-normative errata page for this specification is located at <http://www.oasis-open.org/committees/sca-j/>.

Notices

Copyright © OASIS® 2005, 2008. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The names "OASIS", [insert specific trademarked names and abbreviations here] are trademarks of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see <http://www.oasis-open.org/who/trademark.php> for above guidance.

Table of Contents

1	Introduction	9
1.1	Terminology	9
1.2	Normative References	9
1.3	Non-Normative References	10
2	Implementation Metadata	11
2.1	Service Metadata	11
2.1.1	@Service	11
2.1.2	Java Semantics of a Remotable Service	11
2.1.3	Java Semantics of a Local Service	11
2.1.4	@Reference	12
2.1.5	@Property	12
2.2	Implementation Scopes: @Scope, @Init, @Destroy	12
2.2.1	Stateless scope	13
2.2.2	Composite scope	13
2.2.3	Conversation scope	13
3	Interface	14
3.1	Java interface element – <interface.java>	14
3.2	@Remotable	15
3.3	@Conversational	15
3.4	@Callback	15
4	Client API	16
4.1	Accessing Services from an SCA Component	16
4.1.1	Using the Component Context API	16
4.2	Accessing Services from non-SCA component implementations	16
4.2.1	ComponentContext	16
5	Error Handling	17
6	Asynchronous and Conversational Programming	18
6.1	@OneWay	18
6.2	Conversational Services	18
6.2.1	ConversationAttributes	18
6.2.2	@EndsConversation	19
6.3	Passing Conversational Services as Parameters	19
6.4	Conversational Client	19
6.5	Conversation Lifetime Summary	20
6.6	Conversation ID	21
6.6.1	Application Specified Conversation IDs	21
6.6.2	Accessing Conversation IDs from Clients	21
6.7	Callbacks	21
6.7.1	Stateful Callbacks	21
6.7.2	Stateless Callbacks	23
6.7.3	Implementing Multiple Bidirectional Interfaces	24
6.7.4	Accessing Callbacks	24
6.7.5	Customizing the Callback	25

6.7.6	Customizing the Callback Identity	25
6.7.7	Bindings for Conversations and Callbacks	26
7	Java API	27
7.1	Component Context	27
7.2	Request Context	28
7.3	CallableReference	29
7.4	ServiceReference	30
7.5	Conversation	30
7.6	ServiceRuntimeException	31
7.7	NoRegisteredCallbackException	31
7.8	ServiceUnavailableException	31
7.9	InvalidServiceException	31
7.10	ConversationEndedException	32
8	Java Annotations	33
8.1	@AllowsPassByReference	33
8.2	@Callback	34
8.3	@ComponentName	35
8.4	@Constructor	35
8.5	@Context	36
8.6	@Conversational	37
8.7	@ConversationAttributes	37
8.8	@ConversationID	38
8.9	@Destroy	39
8.10	@EagerInit	40
8.11	@EndsConversation	40
8.12	@Init	41
8.13	@OneWay	41
8.14	@Property	42
8.15	@Reference	43
8.15.1	Reinjection	46
8.16	@Remotable	47
8.17	@Scope	49
8.18	@Service	49
9	WSDL to Java and Java to WSDL	51
9.1	JAX-WS Client Asynchronous API for a Synchronous Service	51
10	Policy Annotations for Java	53
10.1	General Intent Annotations	53
10.2	Specific Intent Annotations	55
10.2.1	How to Create Specific Intent Annotations	56
10.3	Application of Intent Annotations	57
10.3.1	Inheritance And Annotation	58
10.4	Relationship of Declarative And Annotated Intents	59
10.5	Policy Set Annotations	60
10.6	Security Policy Annotations	60
10.6.1	Security Interaction Policy	60

10.6.2 Security Implementation Policy	63
A. XML Schema: sca-interface-java.xsd	67
B. Acknowledgements	68
C. Non-Normative Text	69
D. Revision History	70
1 Introduction	7
1.1 Terminology	7
1.2 Normative References	7
1.3 Non-Normative References	8
2 Implementation Metadata	9
2.1 Service Metadata	9
2.1.1 @Service	9
2.1.2 Java Semantics of a Remotable Service	9
2.1.3 Java Semantics of a Local Service	9
2.1.4 @Reference	10
2.1.5 @Property	10
2.2 Implementation Scopes: @Scope, @Init, @Destroy	10
2.2.1 Stateless scope	11
2.2.2 Composite scope	11
2.2.3 Conversation scope	11
3 Interface	12
3.1 Java interface element ("interface.java")	12
3.2 @Remotable	12
3.3 @Conversational	12
3.4 @Callback	12
4 Client API	13
4.1 Accessing Services from an SCA Component	13
4.1.1 Using the Component Context API	13
4.2 Accessing Services from non-SCA component implementations	13
4.2.1 ComponentContext	13
5 Error Handling	14
6 Asynchronous and Conversational Programming	15
6.1 @OneWay	15
6.2 Conversational Services	15
6.2.1 ConversationAttributes	15
6.2.2 @EndsConversation	16
6.3 Passing Conversational Services as Parameters	16
6.4 Conversational Client	16
6.5 Conversation Lifetime Summary	17
6.6 Conversation ID	18
6.6.1 Application Specified Conversation IDs	18
6.6.2 Accessing Conversation IDs from Clients	18
6.7 Callbacks	18
6.7.1 Stateful Callbacks	18
6.7.2 Stateless Callbacks	20

6.7.3	Implementing Multiple Bidirectional Interfaces	21
6.7.4	Accessing Callbacks	21
6.7.5	Customizing the Callback	22
6.7.6	Customizing the Callback Identity	22
6.7.7	Bindings for Conversations and Callbacks	23
7	Java API	24
7.1	Component Context	24
7.2	Request Context	25
7.3	CallableReference	26
7.4	ServiceReference	26
7.5	Conversation	27
7.6	ServiceRuntimeException	27
7.7	NoRegisteredCallbackException	28
7.8	ServiceUnavailableException	28
7.9	InvalidServiceException	28
7.10	ConversationEndedException	28
8	Java Annotations	30
8.1	@AllowsPassByReference	30
8.2	@Callback	30
8.3	@ComponentName	32
8.4	@Constructor	32
8.5	@Context	33
8.6	@Conversational	34
8.7	@ConversationAttributes	34
8.8	@ConversationID	35
8.9	@Destroy	36
8.10	@EagerInit	36
8.11	@EndsConversation	37
8.12	@Init	37
8.13	@OneWay	38
8.14	@Property	39
8.15	@Reference	40
8.15.1	Reinjection	43
8.16	@Remotable	44
8.17	@Scope	45
8.18	@Service	46
9	WSDL to Java and Java to WSDL	48
9.1	JAX-WS Client Asynchronous API for a Synchronous Service	48
10	Policy Annotations for Java	50
10.1	General Intent Annotations	50
10.2	Specific Intent Annotations	52
10.2.1	How to Create Specific Intent Annotations	53
10.3	Application of Intent Annotations	54
10.3.1	Inheritance And Annotation	55
10.4	Relationship of Declarative And Annotated Intents	56

10.5 Policy Set Annotations.....	57
10.6 Security Policy Annotations.....	57
10.6.1 Security Interaction Policy.....	57
10.6.2 Security Implementation Policy.....	60
A. XML Schema: sca-interface-java.xsd.....	64
B. Acknowledgements.....	65
C. Non Normative Text.....	66
D. Revision History.....	67

1 Introduction

The SCA Common Annotation, APIs, Client and Implementation Model specifies a Java syntax for programming concepts defined in the SCA Assembly Model Specification [ASSEMBLY]. It specifies a set of APIs and annotations that may be used by Java-based SCA specifications.

Specifically, this specification covers:

1. Implementation metadata for specifying component services, references, and properties
2. A client and component API
3. Metadata for asynchronous and conversational services
4. Metadata for callbacks
5. Definitions of standard component implementation scopes
6. Java to WSDL and WSDL to Java mappings
7. Security policy annotations

Note that individual programming models may chose to implement their own mappings of assembly model concepts using native APIs and idioms when appropriate.

The goal of specifying the annotations, APIs, client and implementation model in this specification is to promote consistency and reduce duplication across various Java-related component implementation type specifications. The annotations, APIs, client and implementation model defined in this specification are designed to be used by other SCA Java-related specifications in either a partial or complete fashion.

This document defines implementation metadata using the annotation capability from Java™ 2 Standard Edition (J2SE) 5. However, SCA also allows service clients and implementations to be written using J2SE 1.4. All metadata that is represented by annotations can also be expressed using a component type side file, as defined in the SCA Assembly Specification [ASSEMBLY].

1.1 Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

1.2 Normative References

- | | |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [RFC2119] | S. Bradner, <i>Key words for use in RFCs to Indicate Requirement Levels</i> , http://www.ietf.org/rfc/rfc2119.txt , IETF RFC 2119, March 1997. |
| [ASSEMBLY] | SCA Assembly Specification, http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf |
| [SDO] | SDO 2.1 Specification, http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf |
| [JAX-B] | JAXB 2.1 Specification, http://www.jcp.org/en/jsr/detail?id=222 |
| [WSDL] | WSDL Specification, WSDL 1.1: http://www.w3.org/TR/wsd1 , WSDL 2.0: http://www.w3.org/TR/wsd120/ |
| [POLICY] | SCA Policy Framework, http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf |
| [JSR-250] | Common Annotation for Java Platform specification (JSR-250), http://www.jcp.org/en/jsr/detail?id=250 |

44 **[JAX-WS]** JAX-WS 2.1 Specification (JSR-224), <http://www.jcp.org/en/jsr/detail?id=224>

45 **1.3 Non-Normative References**

46 **TBD** TBD

47 2 Implementation Metadata

48 This section describes SCA Java-based metadata, which applies to Java-based implementation
49 types.

50 2.1 Service Metadata

51 2.1.1 @Service

52
53 The **@Service annotation** is used on a Java class to specify the interfaces of the services
54 implemented by the implementation. Service interfaces are defined in one of the following ways:

- 55 • As a Java interface
- 56 • As a Java class
- 57 • As a Java interface generated from a Web Services Description Language [WSDL]
58 (WSDL) portType (Java interfaces generated from a WSDL portType are always
59 **remotable**)

60 2.1.2 Java Semantics of a Remotable Service

61 A **remotable service** is defined using the @Remotable annotation on the Java interface that
62 defines the service. Remotable services are intended to be used for **coarse grained** services, and
63 the parameters are passed **by-value**. Remotable Services are not allowed to make use of method
64 **overloading**.

65 The following snippet shows an example of a Java interface for a remote service:

```
66 package services.hello;  
67 @Remotable  
68 public interface HelloService {  
69     String hello(String message);  
70 }  
71
```

72 2.1.3 Java Semantics of a Local Service

73 A **local service** can only be called by clients that are deployed within the same address space as
74 the component implementing the local service.

75 A local interface is defined by a Java interface with no @Remotable annotation or it is defined by a
76 Java class.

77 The following snippet shows an example of a Java interface for a local service:

```
78  
79 package services.hello;  
80 public interface HelloService {  
81     String hello(String message);  
82 }  
83
```

84 The style of local interfaces is typically **fine grained** and is intended for **tightly coupled**
85 interactions.

86 The data exchange semantic for calls to local services is **by-reference**. This means that code must
87 be written with the knowledge that changes made to parameters (other than simple types) by
88 either the client or the provider of the service are visible to the other.

89 2.1.4 @Reference

90 Accessing a service using reference injection is done by defining a field, a setter method
91 parameter, or a constructor parameter typed by the service interface and annotated with a
92 **@Reference** annotation.

93 2.1.5 @Property

94 Implementations can be configured with data values through the use of properties, as defined in
95 the SCA Assembly specification [ASSEMBLY]. The **@Property** annotation is used to define an SCA
96 property.

97 2.2 Implementation Scopes: @Scope, @Init, @Destroy

98 Component implementations can either manage their own state or allow the SCA runtime to do so.
99 In the latter case, SCA defines the concept of **implementation scope**, which specifies a visibility
100 and lifecycle contract an implementation has with the SCA runtime. Invocations on a service
101 offered by a component will be dispatched by the SCA runtime to an **implementation instance**
102 according to the semantics of its implementation scope.

103 Scopes are specified using the **@Scope** annotation on the implementation class.

104 This document defines three scopes:

- 105 • STATELESS
- 106 • CONVERSATION
- 107 • COMPOSITE

108 Java-based implementation types can choose to support any of these scopes, and they may define
109 new scopes specific to their type.

110 An implementation type may allow component implementations to declare **lifecycle methods** that
111 are called when an implementation is instantiated or the scope is expired.

112 **@Init** denotes a method called upon first use of an instance during the lifetime of the scope
113 (except for composite scoped implementation marked to eagerly initialize, see [section Composite](#)
114 [Scope](#)).

115 **@Destroy** specifies a method called when the scope ends.

116 Note that only no argument methods with a void return type can be annotated as lifecycle
117 methods.

118 The following snippet is an example showing a fragment of a service implementation annotated
119 with lifecycle methods:

```
120  
121     @Init  
122     public void start() {  
123         ...  
124     }  
125  
126     @Destroy  
127     public void stop() {  
128         ...
```

129 }
130

131 The following sections specify four standard scopes, which a Java-based implementation type may
132 support.

133 **2.2.1 Stateless scope**

134 For stateless scope components, there is no implied correlation between implementation instances
135 used to dispatch service requests.

136 The concurrency model for the stateless scope is single threaded. This means that the SCA
137 runtime **MUST** ensure that a stateless scoped implementation instance object is only ever
138 dispatched on one thread at any one time. In addition, within the SCA lifecycle of an instance, the
139 SCA runtime **MUST** only make a single invocation of one business method. Note that the SCA
140 lifecycle might not correspond to the Java object lifecycle due to runtime techniques such as
141 pooling.

142 **2.2.2 Composite scope**

143 All service requests are dispatched to the same implementation instance for the lifetime of the
144 containing composite. The lifetime of the containing composite is defined as the time it becomes
145 active in the runtime to the time it is deactivated, either normally or abnormally.

146 A composite scoped implementation may also specify eager initialization using the **@EagerInit**
147 annotation. When marked for eager initialization, the composite scoped instance is created when
148 its containing component is started. If a method is marked with the **@Init** annotation, it is called
149 when the instance is created.

150 The concurrency model for the composite scope is multi-threaded. This means that the SCA
151 runtime **MAY** run multiple threads in a single composite scoped implementation instance object
152 and it **MUST NOT** perform any synchronization.

153 **2.2.3 Conversation scope**

154 A **conversation** is defined as a series of correlated interactions between a client and a target
155 service. A conversational scope starts when the first service request is dispatched to an
156 implementation instance offering a conversational service. A conversational scope completes after
157 an end operation defined by the service contract is called and completes processing or the
158 conversation expires. A conversation may be long-running (for example, hours, days or weeks)
159 and the SCA runtime may choose to passivate implementation instances. If this occurs, the
160 runtime must guarantee that implementation instance state is preserved.

161 Note that in the case where a conversational service is implemented by a Java class marked as
162 conversation scoped, the SCA runtime will transparently handle implementation state. It is also
163 possible for an implementation to manage its own state. For example, a Java class having a
164 stateless (or other) scope could implement a conversational service.

165 A conversational scoped class **MUST NOT** expose a service using a non-conversational interface.
166 When a service has a conversational interface it **MUST** be implemented by a conversation-scoped
167 component. If no scope is specified on the implementation, then conversation scope is implied.

168 The concurrency model for the conversation scope is multi-threaded. This means that the SCA
169 runtime **MAY** run multiple threads in a single conversational scoped implementation instance
170 object and it **MUST NOT** perform any synchronization.

171 3 Interface

172 This section describes the SCA Java interface element and the SCA metadata for Java interfaces.

173 3.1 Java interface element `<interface.java>`

174 The Java interface element is used in SCDL files in places where an interface is declared in terms
175 of a Java interface class. The Java interface element identifies the Java interface class and
176 optionally identifies a callback interface, where the first Java interface represents the forward
177 (service) call interface and the second interface represents the interface used to call back from the
178 service to the client.

179

180 The following is the pseudo-schema for the interface.java element~~The following snippet shows the~~
181 schema for the Java interface element.

182

```
183 <interface.java interface="NCName" callbackInterface="NCName"?... />
```

184

185 The interface.java element has the following attributes:

- 186 • **interface (1..1)** – MUST be the fully qualified name of the Java interface class [JCA30001]
- 187 • **callbackInterface (0..1)** – MUST be the fully qualified name of a Java interface used for
188 callbacks [JCA30002]

189

190 The following snippet shows an example of the Java interface element:

191

```
192 <interface.java interface="services.stockquote.StockQuoteService"  
193 callbackInterface="services.stockquote.StockQuoteServiceCallback"/>
```

194

195 Here, the Java interface is defined in the Java class file
196 `./services/stockquote/StockQuoteService.class`, where the root directory is defined by the
197 contribution in which the interface exists. Similarly, the callback interface is defined in the Java
198 class file `./services/stockquote/StockQuoteServiceCallback.class`.

199 Note that the Java interface class identified by the @interface attribute can contain a Java
200 @Callback annotation which identifies a callback interface. If this is the case, then it is not
201 necessary to provide the @callbackInterface attribute. However, if the Java interface class
202 identified by the @interface attribute does contain a Java @Callback annotation, then the Java
203 interface class identified by the @callbackInterface attribute MUST be the same interface class.
204 [JCA30003]

205 For the Java interface type system, arguments and return values of the service methods are
206 described using Java classes or simple Java types. It is recommended that the Java Classes used
207 conform to the requirements of either JAXB [JAXB] or of Service Data Objects [SDO] because of
208 their integration with XML technologies.

209 For the Java interface type system, **parameters and return types** of the service methods are
210 described using Java classes or simple Java types. Service Data Objects [SDO] are the preferred
211 form of Java class because of their integration with XML technologies.

212

213 3.2 @Remotable

214 The **@Remotable** annotation on a Java interface indicates that the interface is designed to be
215 used for remote communication. Remotable interfaces are intended to be used for **coarse**
216 **grained** services. Operations' parameters and return values are passed **by-value**. Remotable
217 Services are not allowed to make use of method **overloading**.

218 3.3 @Conversational

219 Java service interfaces may be annotated to specify whether their contract is conversational as
220 described in the Assembly Specification [ASSEMBLY] by using the **@Conversational** annotation. A
221 conversational service indicates that requests to the service are correlated in some way.

222 When @Conversational is not specified on a service interface, the service contract is **stateless**.

223 3.4 @Callback

224 A callback interface is declared by using a **@Callback** annotation on a Java service interface, with
225 the Java Class object of the callback interface as a parameter. There is another form of the
226 **@Callback** annotation, without any parameters, that specifies callback injection for a setter method
227 or a field of an implementation.

228 4 Client API

229 This section describes how SCA services may be programmatically accessed from components and
230 also from non-managed code, i.e. code not running as an SCA component.

231 4.1 Accessing Services from an SCA Component

232 An SCA component may obtain a service reference either through injection or programmatically
233 through the **ComponentContext** API. Using reference injection is the recommended way to
234 access a service, since it results in code with minimal use of middleware APIs. The
235 ComponentContext API is provided for use in cases where reference injection is not possible.

236 4.1.1 Using the Component Context API

237 When a component implementation needs access to a service where the reference to the service is
238 not known at compile time, the reference can be located using the component's
239 ComponentContext.

240 4.2 Accessing Services from non-SCA component implementations

241 This section describes how Java code not running as an SCA component that is part of an SCA
242 composite accesses SCA services via references.

243 4.2.1 ComponentContext

244 Non-SCA client code can use the ComponentContext API to perform operations against a
245 component in an SCA domain. How client code obtains a reference to a ComponentContext is
246 runtime specific.

247 The following example demonstrates the use of the component Context API by non-SCA code:

248

```
249 ComponentContext context = // obtained through host environment-specific means  
250 HelloService helloService =  
251     context.getService(HelloService.class, "HelloService");  
252 String result = helloService.hello("Hello World!");
```

253 5 Error Handling

254 Clients calling service methods may experience business exceptions and SCA runtime exceptions.

255 Business exceptions are thrown by the implementation of the called service method, and are
256 defined as checked exceptions on the interface that types the service.

257 SCA runtime exceptions are raised by the SCA runtime and signal problems in management of
258 component execution or problems interacting with remote services. The SCA runtime exceptions
259 are [defined in the Java API section](#).

260 6 Asynchronous and Conversational Programming

261 Asynchronous programming of a service is where a client invokes a service and carries on
262 executing without waiting for the service to execute. Typically, the invoked service executes at
263 some later time. Output from the invoked service, if any, must be fed back to the client through a
264 separate mechanism, since no output is available at the point where the service is invoked. This is
265 in contrast to the call-and-return style of synchronous programming, where the invoked service
266 executes and returns any output to the client before the client continues. The SCA asynchronous
267 programming model consists of:

- 268 • support for non-blocking method calls
- 269 • conversational services
- 270 • callbacks

271 Each of these topics is discussed in the following sections.

272 Conversational services are services where there is an ongoing sequence of interactions between
273 the client and the service provider, which involve some set of state data – in contrast to the
274 simple case of stateless interactions between a client and a provider. Asynchronous services may
275 often involve the use of a conversation, although this is not mandatory.

276 6.1 @OneWay

277 **Nonblocking calls** represent the simplest form of asynchronous programming, where the client of
278 the service invokes the service and continues processing immediately, without waiting for the
279 service to execute.

280 Any method with a void return type and has no declared exceptions may be marked with a
281 **@OneWay** annotation. This means that the method is non-blocking and communication with the
282 service provider may use a binding that buffers the requests and sends it at some later time.

283 For a Java client to make a non-blocking call to methods that either return values or which throw
284 exceptions, a Java client can use the JAX-WS asynchronous client API model that is described in
285 section 9. It is considered to be a best practice that service designers define one-way methods as
286 often as possible, in order to give the greatest degree of binding flexibility to deployers.

287 6.2 Conversational Services

288 A service may be declared as conversational by marking its Java interface with a
289 **@Conversational** annotation. If a service interface is not marked with a **@Conversational**, it is
290 stateless.

291 6.2.1 ConversationAttributes

292 A Java-based implementation class may be marked with a **@ConversationAttributes** annotation,
293 which is used to specify the expiration rules for conversational implementation instances.

294 An example of the **@ConversationAttributes** is shown below:

```
295 package com.bigbank;  
296 import org.eseaopen.sca.annotations.ConversationAttributes;  
297  
298 @ConversationAttributes(maxAge="30 days");  
299 public class LoanServiceImpl implements LoanService {  
300  
301 }
```

302 6.2.2 @EndsConversation

303 A method of a conversational interface may be marked with an @EndsConversation annotation.
304 Once a method marked with @EndsConversation has been called, the conversation between client
305 and service provider is at an end, which implies no further methods may be called on that service
306 within the same conversation. This enables both the client and the service provider to free up
307 resources that were associated with the conversation.

308 It is also possible to mark a method on a callback interface (described later) with
309 @EndsConversation, in order for the service provider to be the party that chooses to end the
310 conversation.

311 If a conversation is ended with an explicit outbound call to an @EndsConversation method or
312 through a call to the ServiceReference.endConversation() method, then any subsequent call to an
313 operation on the service reference will start a new conversation. If the conversation ends for any
314 other reason (e.g. a timeout occurred), then until ServiceReference.getConversation().end() is
315 called, the ConversationEndedException is thrown by any conversational operation.

316 6.3 Passing Conversational Services as Parameters

317 The service reference which represents a single conversation can be passed as a parameter to
318 another service, even if that other service is remote. This may be used to allow one component to
319 continue a conversation that had been started by another.

320 A service provider may also create a service reference for itself that it can pass to other services.
321 A service implementation does this with a call to the createSelfReference(...) method:

```
322     interface ComponentContext{  
323         ...  
324         <B> ServiceReference<B> createSelfReference(Class  
325             businessInterface);  
326         <B> ServiceReference<B> createSelfReference(Class  
327             businessInterface, String serviceName);  
328     }
```

329
330 The second variant, which takes an additional **serviceName** parameter, must be used if the
331 component implements multiple services.

332 This capability may be used to support complex callback patterns, such as when a callback is
333 applicable only to a subset of a larger conversation. Simple callback patterns are handled by the
334 built-in callback support described later.

335 6.4 Conversational Client

336 The client of a conversational service does not need to be coded in a special way. The client can
337 take advantage of the conversational nature of the interface through the relationship of the
338 different methods in the interface and any data they may share in common. If the service is
339 asynchronous, the client may like to use a feature such as the conversationID to keep track of any
340 state data relating to the conversation.

341 The developer of the client knows that the service is conversational by introspecting the service
342 contract. The following shows how a client accesses the conversational service described above:

```
343  
344     @Reference  
345     LoanService loanService;  
346     // Known to be conversational because the interface is marked as  
347     // conversational
```

```

348     public void applyForMortgage(Customer customer, HouseInfo houseInfo,
349                                 int term)
350     {
351         LoanApplication loanApp;
352         loanApp = createApplication(customer, houseInfo);
353         loanService.apply(loanApp);
354         loanService.lockCurrentRate(term);
355     }
356
357     public boolean isApproved() {
358         return loanService.getLoanStatus().equals("approved");
359     }
360     public LoanApplication createApplication(Customer customer,
361                                             HouseInfo houseInfo) {
362         return ...;
363     }

```

364 6.5 Conversation Lifetime Summary

365 **Starting conversations**

366 Conversations start on the client side when one of the following occur:

- 367 • A @Reference to a conversational service is injected
- 368 • A call is made to CompositeContext.getServiceReference and then a method of the service
- 369 is called.
- 370

371 **Continuing conversations**

372 The client can continue an existing conversation, by:

- 373 • Holding the service reference that was created when the conversation started
- 374 • Getting the service reference object passed as a parameter from another service, even
- 375 remotely
- 376 • Loading a service reference that had been written to some form of persistent storage
- 377

378 **Ending conversations**

379 A conversation ends, and any state associated with the conversation is freed up, when:

- 380 • A service operation that has been annotated @EndsConversation has been called
- 381 • The server calls an @EndsConversation method on the @Callback reference
- 382 • The server's conversation lifetime timeout occurs
- 383 • The client calls Conversation.end()
- 384 • Any non-business exception is thrown by a conversational operation
- 385

386 If a method is invoked on a service reference after an @EndsConversation method has been called
387 then a new conversation will automatically be started. If
388 ServiceReference.getConversationID() is called after the @EndsConversation method is called,
389 but before the next conversation has been started, it returns null.

390 If a service reference is used after the service provider's conversation timeout has caused the
391 conversation to be ended, then ConversationEndedException is thrown. In order to use that
392 service reference for a new conversation, its endConversation () method must be called.
393

394 6.6 Conversation ID

395 Every conversation has a **conversation ID**. The conversation ID can be generated by the system,
396 or it can be supplied by the client component.

397 If a field or setter method is annotated with **@ConversationID**, then the conversation ID for the
398 conversation is injected. The type of the field is not necessarily String. System generated
399 conversation IDs are always strings, but application generated conversation IDs may be other
400 complex types.

401 6.6.1 Application Specified Conversation IDs

402 It is possible to take advantage of the state management aspects of conversational services while
403 using a client-provided conversation ID. To do this, the client does not use reference injection,
404 but uses the **ServiceReference.setConversationID()** API.

405 The conversation ID that is passed into this method should be an instance of either a String or of
406 an object that is serializable into XML. The ID must be unique to the client component over all
407 time. If the client is not an SCA component, then the ID must be globally unique.

408 Not all conversational service bindings support application-specified conversation IDs or may only
409 support application-specified conversation IDs that are Strings.

410 6.6.2 Accessing Conversation IDs from Clients

411 Whether the conversation ID is chosen by the client or is generated by the system, the client may
412 access the conversation ID by calling **getConversationID()** on the current conversation
413 object.

414 If the conversation ID is not application specified, then the
415 **ServiceReference.getConversationID()** method is only guaranteed to return a valid value
416 after the first operation has been invoked, otherwise it returns null.

417 6.7 Callbacks

418 A **callback service** is a service that is used for **asynchronous** communication from a service
419 provider back to its client, in contrast to the communication through return values from
420 synchronous operations. Callbacks are used by **bidirectional services**, which are services that
421 have two interfaces:

- 422 • an interface for the provided service
- 423 • a callback interface that must be provided by the client

424 Callbacks may be used for both remotable and local services. Either both interfaces of a
425 bidirectional service must be remotable, or both must be local. It is illegal to mix the two. There
426 are two basic forms of callbacks: stateless callbacks and stateful callbacks.

427 A callback interface is declared by using a **@Callback** annotation on a service interface, with the
428 Java Class object of the interface as a parameter. The annotation may also be applied to a method
429 or to a field of an implementation, which is used in order to have a callback injected, as explained
430 in the next section.

431 6.7.1 Stateful Callbacks

432 A **stateful** callback represents a specific implementation instance of the component that is the
433 client of the service. The interface of a stateful callback should be marked as **conversational**.

434 The following example interfaces show an interaction over a stateful callback.

```
435 package somepackage;
436 import org.esea.oasisopen.sca.annotations.Callback;
437 import org.esea.oasisopen.sca.annotations.Conversational;
438 import org.esea.oasisopen.sca.annotations.Remotable;
439 @Remotable
440 @Conversational
441 @Callback(MyServiceCallback.class)
442 public interface MyService {
443
444     void someMethod(String arg);
445 }
446
447 @Remotable
448 @Conversational
449 public interface MyServiceCallback {
450
451     void receiveResult(String result);
452 }
453
```

454 An implementation of the service in this example could use the @Callback annotation to request
455 that a stateful callback be injected. The following is a fragment of an implementation of the
456 example service. In this example, the request is passed on to some other component, so that the
457 example service acts essentially as an intermediary. If the example service is conversation
458 scoped, the callback will still be available when the backend service sends back its asynchronous
459 response.

460 When an interface and its callback interface are both marked as conversational, then there is only
461 one conversation that applies in both directions and it has the same lifetime. In this case, if both
462 interfaces declare a @ConversationAttributes annotation, then only the annotation on the main
463 interface applies.

```
464 @Callback
465 protected MyServiceCallback callback;
466
467 @Reference
468 protected MyService backendService;
469
470
471 public void someMethod(String arg) {
472     backendService.someMethod(arg);
473 }
474
475 public void receiveResult(String result) {
476     callback.receiveResult(result);
477 }
478
```

479 This fragment must come from an implementation that offers two services, one that it offers to its
480 clients (MyService) and one that is used for receiving callbacks from the back end
481 (MyServiceCallback). The code snippet below is taken from the client of this service, which also
482 implements the methods defined in MyServiceCallback.

483

```

484
485     private MyService myService;
486
487     @Reference
488     public void setMyService(MyService service) {
489         myService = service;
490     }
491
492     public void aClientMethod() {
493         ...
494         myService.someMethod(arg);
495     }
496
497     public void receiveResult(String result) {
498         // code to process the result
499     }
500

```

501 Stateful callbacks support some of the same use cases as are supported by the ability to pass
502 service references as parameters. The primary difference is that stateful callbacks do not require
503 any additional parameters be passed with service operations. This can be a great convenience. If
504 the service has many operations and any of those operations could be the first operation of the
505 conversation, it would be unwieldy to have to take a callback parameter as part of every
506 operation, just in case it is the first operation of the conversation. It is also more natural than
507 requiring application developers to invoke an explicit operation whose only purpose is to pass the
508 callback object that should be used.

509 6.7.2 Stateless Callbacks

510 A stateless callback interface is a callback whose interface is not marked as **conversational**.
511 Unlike stateful services, a client that uses stateless callbacks will not have callback methods
512 routed to an instance of the client that contains any state that is relevant to the conversation. As
513 such, it is the responsibility of such a client to perform any persistent state management itself.
514 The only information that the client has to work with (other than the parameters of the callback
515 method) is a callback ID object that is passed with requests to the service and is guaranteed to be
516 returned with any callback.

517 The following is a repeat of the client code fragment above, but with the assumption that in this
518 case the MyServiceCallback is stateless. The client in this case needs to set the callback ID before
519 invoking the service and then needs to get the callback ID when the response is received.

```

520
521     private ServiceReference<MyService> myService;
522
523     @Reference
524     public void setMyService(ServiceReference<MyService> service) {
525         myService = service;
526     }
527
528     public void aClientMethod() {
529         String someKey = "1234";
530         ...
531
532         myService.setCallbackID(someKey);
533         myService.getService().someMethod(arg);
534     }
535
536     @Context RequestContext context;
537
538     public void receiveResult(String result) {

```

```
539     Object key = context.getServiceReference().getCallbackID();
540     // Lookup any relevant state based on "key"
541     // code to process the result
542 }
```

543

544 Just as with stateful callbacks, a service implementation gets access to the callback object by
545 annotating a field or setter method with the `@Callback` annotation, such as the following:

```
546 @Callback
547 protected MyServiceCallback callback;
```

549

550 The difference for stateless services is that the callback field would not be available if the
551 component is servicing a request for anything other than the original client. So, the technique
552 used in the previous section, where there was a response from the backendService which was
553 forwarded as a callback from MyService would not work because the callback field would be null
554 when the message from the backend system was received.

555 6.7.3 Implementing Multiple Bidirectional Interfaces

556 Since it is possible for a single implementation class to implement multiple services, it is also
557 possible for callbacks to be defined for each of the services that it implements. The service
558 implementation can include an injected field for each of its callbacks. The runtime injects the
559 callback onto the appropriate field based on the type of the callback. The following shows the
560 declaration of two fields, each of which corresponds to a particular service offered by the
561 implementation.

```
562 @Callback
563 protected MyService1Callback callback1;
564
565 @Callback
566 protected MyService2Callback callback2;
```

568

569 If a single callback has a type that is compatible with multiple declared callback fields, then all of
570 them will be set.

571 6.7.4 Accessing Callbacks

572 In addition to injecting a reference to a callback service, it is also possible to obtain a reference to
573 a Callback instance by annotating a field or method with the `@Callback` annotation.

574

575 A reference implementing the callback service interface may be obtained using
576 `CallableReference.getService()`.

577 The following example fragments come from a service implementation that uses the callback API:

```
578 @Callback
579 protected CallableReference<MyCallback> callback;
580
581 public void someMethod() {
582     MyCallback myCallback = callback.getCallback();    ...
583     myCallback.receiveResult(theResult);
584 }
585
586
587
588
```


589 Alternatively, a callback may be retrieved programmatically using the **RequestContext** API. The
590 snippet below shows how to retrieve a callback in a method programmatically:

591

```
592 public void someMethod() {  
593     MyCallback myCallback =  
594         ComponentContext.getRequestContext().getCallback();  
595     ...  
596     ...  
597     myCallback.receiveResult(theResult);  
598 }  
599  
600  
601
```

602 On the client side, the service that implements the callback can access the callback ID that was
603 returned with the callback operation by accessing the request context, as follows:

604

```
605 @Context  
606 protected RequestContext requestContext;  
607  
608 void receiveResult(Object theResult) {  
609     Object refParams =  
610         requestContext.getServiceReference().getCallbackID();  
611     ...  
612 }  
613
```

614

615 On the client side, the object returned by the `getServiceReference()` method represents the
616 service reference for the callback. The object returned by `getCallbackID()` represents the
617 identity associated with the callback, which may be a single String or may be an object (as
618 described below in "Customizing the Callback Identity").

619 6.7.5 Customizing the Callback

620 By default, the client component of a service is assumed to be the callback service for the
621 bidirectional service. However, it is possible to change the callback by using the
622 **ServiceReference.setCallback()** method. The object passed as the callback should implement
623 the interface defined for the callback, including any additional SCA semantics on that interface
624 such as whether or not it is remotable.

625 Since a service other than the client can be used as the callback implementation, SCA does not
626 generate a deployment-time error if a client does not implement the callback interface of one of its
627 references. However, if a call is made on such a reference without the `setCallback()` method
628 having been called, then a **NoRegisteredCallbackException** is thrown on the client.

629 A callback object for a stateful callback interface has the additional requirement that it must be
630 serializable. The SCA runtime may serialize a callback object and persistently store it.

631 A callback object may be a service reference to another service. In that case, the callback
632 messages go directly to the service that has been set as the callback. If the callback object is not
633 a service reference, then callback messages go to the client and are then routed to the specific
634 instance that has been registered as the callback object. However, if the callback interface has a
635 stateless scope, then the callback object **must** be a service reference.

636 6.7.6 Customizing the Callback Identity

637 The identity that is used to identify a callback request is initially generated by the system.
638 However, it is possible to provide an application specified identity to identify the callback by calling

639 the **ServiceReference.setCallbackID()** method. This can be used both for stateful and for
640 stateless callbacks. The identity is sent to the service provider, and the binding must guarantee
641 that the service provider will send the ID back when any callback method is invoked.

642 The callback identity has the same restrictions as the conversation ID. It should either be a string
643 or an object that can be serialized into XML. Bindings determine the particular mechanisms to use
644 for transmission of the identity and these may lead to further restrictions when using a given
645 binding.

646 **6.7.7 Bindings for Conversations and Callbacks**

647 There are potentially many ways of representing the conversation ID for conversational services
648 depending on the type of binding that is used. For example, it may be possible WS-RM sequence
649 ids for the conversation ID if reliable messaging is used in a Web services binding. WS-Eventing
650 uses a different technique (the wse:Identity header). There is also a WS-Context OASIS TC that
651 is creating a general purpose mechanism for exactly this purpose.

652 SCA's programming model supports conversations, but it leaves up to the binding the means by
653 which the conversation ID is represented on the wire.

654 7 Java API

655 This section provides a reference for the Java API offered by SCA.

656 7.1 Component Context

657 The following Java code defines the **ComponentContext** interface:

658

```
659 package org.oasisopen.sca;
660
661 public interface ComponentContext {
662     String getURI();
663
664     <B> B getService(Class<B> businessInterface, String referenceName);
665
666     <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
667                                             String referenceName);
668
669     <B> Collection<B> getServices(Class<B> businessInterface,
670                               String referenceName);
671
672     <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
673                                                         businessInterface, String referenceName);
674
675     <B> ServiceReference<B> createSelfReference(Class<B>
676                                               businessInterface);
677
678     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
679                                               String serviceName);
680
681     <B> B getProperty(Class<B> type, String propertyName);
682
683     <B, R extends CallableReference<B>> R cast(B target)
684                                     throws IllegalArgumentException;
685
686     RequestContext getRequestContext();
687
688
689 }
```

690

- 691 • **getURI()** - returns the absolute URI of the component within the SCA domain
- 692 • **getService(Class businessInterface, String referenceName)** – Returns a proxy for
693 the reference defined by the current component. The getService() method takes as its
694 input arguments the Java type used to represent the target service on the client and the
695 name of the service reference. It returns an object providing access to the service. The
696 returned object implements the Java interface the service is typed with. This method
697 MUST throw an IllegalArgumentException if the reference has multiplicity greater than
698 one.
- 699 • **getServiceReference(Class businessInterface, String referenceName)** – Returns a
700 ServiceReference defined by the current component. This method MUST throw an
701 IllegalArgumentException if the reference has multiplicity greater than one.

- 702 • **getServices(Class businessInterface, String referenceName)** – Returns a list of
703 typed service proxies for a business interface type and a reference name.
- 704 • **getServiceReferences(Class businessInterface, String referenceName)** –Returns a
705 list typed service references for a business interface type and a reference name.
- 706 • **createSelfReference(Class businessInterface)** – Returns a ServiceReference that can
707 be used to invoke this component over the designated service.
- 708 • **createSelfReference(Class businessInterface, String serviceName)** – Returns a
709 ServiceReference that can be used to invoke this component over the designated service.
710 Service name explicitly declares the service name to invoke
- 711 • **getProperty (Class type, String propertyName)** - Returns the value of an SCA
712 property defined by this component.
- 713 • **getRequestContext()** - Returns the context for the current SCA service request, or null if
714 there is no current request or if the context is unavailable. This method MUST return non-
715 null when invoked during the execution of a Java business method for a service operation
716 or callback operation, on the same thread that the SCA runtime provided, and MUST
717 return null in all other cases.
- 718 • **cast(B target)** - Casts a type-safe reference to a CallableReference

719 A component may access its component context by defining a field or setter method typed by
720 **org.eseaopen.sca.ComponentContext** and annotated with **@Context**. To access the target
721 service, the component uses **ComponentContext.getService(..)**.

722
723 The following shows an example of component context usage in a Java class using the @Context
724 annotation.

```
725 private ComponentContext componentContext;
726
727 @Context
728 public void setContext(ComponentContext context) {
729     componentContext = context;
730 }
731
732 public void doSomething() {
733     HelloWorld service =
734     componentContext.getService(HelloWorld.class, "HelloWorldComponent");
735     service.hello("hello");
736 }
737
```

738 Similarly, non-SCA client code can use the ComponentContext API to perform operations against a
739 component in an SCA domain. How the non-SCA client code obtains a reference to a
740 ComponentContext is runtime specific.

741 7.2 Request Context

742 The following shows the **RequestContext** interface:

```
743
744 package org.eseaopen.sca;
745
746 import javax.security.auth.Subject;
747
748 public interface RequestContext {
749
750     Subject getSecuritySubject();
751
```

```

752     String getServiceName();
753     <CB> CallableReference<CB> getCallbackReference();
754     <CB> CB getCallback();
755     <B> CallableReference<B> getServiceReference();
756
757 }
758

```

759 The RequestContext interface has the following methods:

- 760 • **getSecuritySubject()** – Returns the JAAS Subject of the current request
- 761 • **getServiceName()** – Returns the name of the service on the Java implementation the
762 request came in on
- 763 • **getCallbackReference()** – Returns a callable reference to the callback as specified by the
764 caller. This method returns null when called for a service request whose interface is not
765 bidirectional or when called for a callback request.
- 766 • **getCallback()** – Returns a proxy for the callback as specified by the caller. Similar to the
767 getCallbackReference() method, this method returns null when called for a service request
768 whose interface is not bidirectional or when called for a callback request.
- 769 • **getServiceReference()** – When invoked during the execution of a service operation, this
770 method MUST return a CallableReference that represents the service that was invoked.
771 When invoked during the execution of a callback operation, this method MUST return a
772 CallableReference that represents the callback that was invoked.

773 7.3 CallableReference

774 The following Java code defines the **CallableReference** interface:

```

775
776 package org.oasisopen.sca;
777
778 public interface CallableReference<B> extends java.io.Serializable {
779
780     B getService();
781     Class<B> getBusinessInterface();
782     boolean isConversational();
783     Conversation getConversation();
784     Object getCallbackID();
785 }
786

```

787 The CallableReference interface has the following methods:

- 788
- 789 • **getService()** - Returns a type-safe reference to the target of this reference. The instance
790 returned is guaranteed to implement the business interface for this reference. The value
791 returned is a proxy to the target that implements the business interface associated with this
792 reference.
- 793 • **getBusinessInterface()** – Returns the Java class for the business interface associated with
794 this reference.
- 795 • **isConversational()** – Returns true if this reference is conversational.
- 796 • **getConversation()** – Returns the conversation associated with this reference. Returns null if
797 no conversation is currently active.
- 798 • **getCallbackID()** – Returns the callback ID.

7.4 ServiceReference

799

800
801 ServiceReferences may be injected using the @Reference annotation on a field, a setter method,
802 or constructor parameter taking the type ServiceReference. The detailed description of the usage
803 of these methods is described in the section on Asynchronous Programming in this document.

804 The following Java code defines the ServiceReference interface:

805

```
806 package org.esoooasisopen.sca;
807
808 public interface ServiceReference<B> extends CallableReference<B> {
809     Object getConversationID();
810     void setConversationID(Object conversationId) throws
811         IllegalStateException;
812     void setCallbackID(Object callbackID);
813     Object getCallback();
814     void setCallback(Object callback);
815 }
816
```

817

818 The ServiceReference interface has the methods of CallableReference plus the following:

819

- 820 • **getConversationID()** - Returns the id supplied by the user that will be associated with
821 future conversations initiated through this reference, or null if no ID has been set by the
822 user.
- 823 • **setConversationID(Object conversationId)** – Set the ID, supplied by the user, to associate
824 with any future conversation started through this reference. If the value supplied is null then
825 the id will be generated by the implementation. Throws an IllegalStateException if a
826 conversation is currently associated with this reference.
- 827 • **setCallbackID(Object callbackID)** – Sets the callback ID.
- 828 • **getCallback()** – Returns the callback object.
- 829 • **setCallback(Object callback)** – Sets the callback object.

7.5 Conversation

831 The following snippet defines Conversation:

832

```
833 package org.esoooasisopen.sca;
834
835 public interface Conversation {
836     Object getConversationID();
837     void end();
838 }
839
```

839

840 The Conversation interface has the following methods:

- 841 • **getConversationID()** – Returns the identifier for this conversation. If a user-defined identity
842 had been supplied for this reference then its value will be returned; otherwise the identity
843 generated by the system when the conversation was initiated will be returned.
- 844 • **end()** – Ends this conversation.

845 7.6 ServiceRuntimeException

846 The following snippet shows the **ServiceRuntimeException**.

847

```
848 package org.esea.oasisopen.sca;
849
850 public class ServiceRuntimeException extends RuntimeException {
851     ...
852 }
853
```

854 This exception signals problems in the management of SCA component execution.

855 7.7 NoRegisteredCallbackException

856 The following snippet shows the **NoRegisteredCallbackException**.

857

```
858 package org.esea.oasisopen.sca;
859
860 public class NoRegisteredCallbackException extends
861     ServiceRuntimeException {
862     ...
863 }
864
```

864 This exception signals a problem where an attempt is made to invoke a callback when a client
865 does not implement the Callback interface and no valid custom Callback has been specified via a
866 call to **ServiceReference.setCallback()**.

867 7.8 ServiceUnavailableException

868 The following snippet shows the **ServiceUnavailableException**.

869

```
870 package org.esea.oasisopen.sca;
871
872 public class ServiceUnavailableException extends ServiceRuntimeException {
873     ...
874 }
875
```

876 This exception signals problems in the interaction with remote services. These are exceptions
877 that may be transient, so retrying is appropriate. Any exception that is a
878 ServiceRuntimeException that is *not* a ServiceUnavailableException is unlikely to be resolved by
879 retrying the operation, since it most likely requires human intervention

880 7.9 InvalidServiceException

881 The following snippet shows the **InvalidServiceException**.

882

```
883 package org.esea.oasisopen.sca;
884
885 public class InvalidServiceException extends ServiceRuntimeException {
886     ...
887 }
888
```

889 This exception signals that the ServiceReference is no longer valid. This can happen when the
890 target of the reference is undeployed. This exception is not transient and therefore is unlikely to
891 be resolved by retrying the operation and will most likely require human intervention.

892 7.10 ConversationEndedException

893 The following snippet shows the *ConversationEndedException*.

```
894 package org.escaoasisopen.sca;
895
896 public class ConversationEndedException extends ServiceRuntimeException {
897     ...
898 }
899
900
```


901

8 Java Annotations

902

This section provides definitions of all the Java annotations which apply to SCA.

903

This specification places constraints on some annotations that are not detectable by a Java compiler. For example, the definition of the `@Property` and `@Reference` annotations indicate that they are allowed on parameters, but sections 8.14 and 8.15 constrain those definitions to constructor parameters. An SCA runtime MUST verify the proper use of all annotations and if an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the invalid implementation code.

904

905

906

907

908

909

SCA annotations are not allowed on static methods and static fields. It is an error to use an SCA annotation on a static method or a static field of an implementation class and the SCA runtime MUST NOT instantiate such an implementation class.

910

911

912

8.1 `@AllowsPassByReference`

913

The following Java code defines the `@AllowsPassByReference` annotation:

914

915

```
package org.eseaoasisopen.sca.annotations;
```

916

917

```
import static java.lang.annotation.ElementType.TYPE;
```

918

```
import static java.lang.annotation.ElementType.METHOD;
```

919

```
import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

920

```
import java.lang.annotation.Retention;
```

921

```
import java.lang.annotation.Target;
```

922

```
@Target({TYPE, METHOD})
```

923

```
@Retention(RUNTIME)
```

924

```
public @interface AllowsPassByReference {
```

925

926

```
}
```

927

928

929

The `@AllowsPassByReference` annotation is used on implementations of remotable interfaces to indicate that interactions with the service from a client within the same address space are allowed to use pass by reference data exchange semantics. The implementation promises that its by-value semantics will be maintained even if the parameters and return values are actually passed by-reference. This means that the service will not modify any operation input parameter or return value, even after returning from the operation. Either a whole class implementing a remotable service or an individual remotable service method implementation can be annotated using the `@AllowsPassByReference` annotation.

930

931

932

933

934

935

936

937

`@AllowsPassByReference` has no attributes

938

939

The following snippet shows a sample where `@AllowsPassByReference` is defined for the implementation of a service method on the Java component implementation class.

940

941

942

```
@AllowsPassByReference
```

943

```
public String hello(String message) {
```

944

```
    ...
```

945

```
}
```

946 8.2 @Callback

947 The following Java code defines shows the **@Callback** annotation:

948

```
949 package org.esea.oasisopen.sca.annotations;
950
951 import static java.lang.annotation.ElementType.TYPE;
952 import static java.lang.annotation.ElementType.METHOD;
953 import static java.lang.annotation.ElementType.FIELD;
954 import static java.lang.annotation.RetentionPolicy.RUNTIME;
955 import java.lang.annotation.Retention;
956 import java.lang.annotation.Target;
957
958 @Target(TYPE, METHOD, FIELD)
959 @Retention(RUNTIME)
960 public @interface Callback {
961     Class<?> value() default Void.class;
962 }
963
964
965
```

966 The @Callback annotation is used to annotate a service interface with a callback interface, which
967 takes the Java Class object of the callback interface as a parameter.

968 The @Callback annotation has the following attribute:

- 969 • **value** – the name of a Java class file containing the callback interface

970

971 The @Callback annotation may also be used to annotate a method or a field of an SCA
972 implementation class, in order to have a callback object injected

973

974 The following snippet shows a @Callback annotation on an interface:

975

```
976 @Remotable
977 @Callback(MyServiceCallback.class)
978 public interface MyService {
979
980     void someAsyncMethod(String arg);
981 }
982
```

983 An example use of the @Callback annotation to declare a callback interface follows:

984

```
985 package somepackage;
986 import org.esea.oasisopen.sca.annotations.Callback;
987 import org.esea.oasisopen.sca.annotations.Remotable;
988 @Remotable
989 @Callback(MyServiceCallback.class)
990 public interface MyService {
991
992     void someMethod(String arg);
993 }
994
995 @Remotable
996 public interface MyServiceCallback {
```

```
997
998     void receiveResult(String result);
999 }
```

1000

1001 In this example, the implied component type is:

1002

```
1003 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >
1004     <service name="MyService">
1005         <interface.java interface="somepackage.MyService"
1006             callbackInterface="somepackage.MyServiceCallback"/>
1007     </service>
1008 </componentType>
```

1010 8.3 @ComponentName

1011 The following Java code defines the **@ComponentName** annotation:

1012

```
1013 package org.esea.oasisopen.sca.annotations;
1014
1015 import static java.lang.annotation.ElementType.METHOD;
1016 import static java.lang.annotation.ElementType.FIELD;
1017 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1018 import java.lang.annotation.Retention;
1019 import java.lang.annotation.Target;
1020
1021 @Target({METHOD, FIELD})
1022 @Retention(RUNTIME)
1023 public @interface ComponentName {
1024
1025 }
1026
```

1027 The @ComponentName annotation is used to denote a Java class field or setter method that is
1028 used to inject the component name.

1029

1030 The following snippet shows a component name field definition sample.

1031

```
1032 @ComponentName
1033 private String componentName;
1034
```

1035 The following snippet shows a component name setter method sample.

1036

```
1037 @ComponentName
1038 public void setComponentName(String name) {
1039     //...
1040 }
```

1041 8.4 @Constructor

1042 The following Java code defines the **@Constructor** annotation:

1043

```

1044 package org.esoasopen.sca.annotations;
1045
1046 import static java.lang.annotation.ElementType.CONSTRUCTOR;
1047 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1048 import java.lang.annotation.Retention;
1049 import java.lang.annotation.Target;
1050
1051 @Target (CONSTRUCTOR)
1052 @Retention (RUNTIME)
1053 public @interface Constructor { }
1054

```

1055 The @Constructor annotation is used to mark a particular constructor to use when instantiating a
1056 Java component implementation. If this constructor has parameters, each of these parameters
1057 MUST have either a @Property annotation or a @Reference annotation.

1058 The following snippet shows a sample for the @Constructor annotation.

```

1059
1060 public class HelloServiceImpl implements HelloService {
1061
1062     public HelloServiceImpl () {
1063         ...
1064     }
1065
1066     @Constructor
1067     public HelloServiceImpl (@Property (name="someProperty") String
1068     someProperty ) {
1069         ...
1070     }
1071
1072     public String hello (String message) {
1073         ...
1074     }
1075 }

```

1076 8.5 @Context

1077 The following Java code defines the **@Context** annotation:

```

1078
1079 package org.esoasopen.sca.annotations;
1080
1081 import static java.lang.annotation.ElementType.METHOD;
1082 import static java.lang.annotation.ElementType.FIELD;
1083 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1084 import java.lang.annotation.Retention;
1085 import java.lang.annotation.Target;
1086
1087 @Target ({METHOD, FIELD})
1088 @Retention (RUNTIME)
1089 public @interface Context {
1090
1091 }
1092

```

1093 The @Context annotation is used to denote a Java class field or a setter method that is used to
1094 inject a composite context for the component. The type of context to be injected is defined by the

1095 type of the Java class field or type of the setter method input argument; the type is either
1096 **ComponentContext** or **RequestContext**.

1097 The @Context annotation has no attributes.

1098

1099 The following snippet shows a ComponentContext field definition sample.

1100

```
1101 @Context  
1102 protected ComponentContext context;  
1103
```

1104 The following snippet shows a RequestContext field definition sample.

1105

```
1106 @Context  
1107 protected RequestContext context;
```

1108 8.6 @Conversational

1109 The following Java code defines the **@Conversational** annotation:

1110

```
1111 package org.esoasisopen.sca.annotations;  
1112  
1113 import static java.lang.annotation.ElementType.TYPE;  
1114 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1115 import java.lang.annotation.Retention;  
1116 import java.lang.annotation.Target;  
1117 @Target (TYPE)  
1118 @Retention (RUNTIME)  
1119 public @interface Conversational {  
1120 }  
1121
```

1122 The @Conversational annotation is used on a Java interface to denote a conversational service
1123 contract.

1124 The @Conversational annotation has no attributes.

1125 The following snippet shows a sample for the @Conversational annotation.

```
1126 package services.hello;  
1127  
1128 import org.esoasisopen.sca.annotations.Conversational;  
1129  
1130 @Conversational  
1131 public interface HelloService {  
1132     void setName (String name);  
1133     String sayHello();  
1134 }
```

1135 8.7 @ConversationAttributes

1136 The following Java code defines the **@ConversationAttributes** annotation:

1137

```
1138 package org.esoasisopen.sca.annotations;  
1139  
1140 import static java.lang.annotation.ElementType.TYPE;  
1141 import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```

1142 import java.lang.annotation.Retention;
1143 import java.lang.annotation.Target;
1144
1145 @Target (TYPE)
1146 @Retention (RUNTIME)
1147 public @interface ConversationAttributes {
1148
1149     String maxIdleTime() default "";
1150     String maxAge() default "";
1151     boolean singlePrincipal() default false;
1152 }
1153

```

1154 The `@ConversationAttributes` annotation is used to define a set of attributes which apply to
1155 conversational interfaces of services or references of a Java class. The annotation has the following
1156 attributes:

- 1157 • ***maxIdleTime (optional)*** - The maximum time that can pass between successive
1158 operations within a single conversation. If more time than this passes, then the container
1159 may end the conversation.
- 1160 • ***maxAge (optional)*** - The maximum time that the entire conversation can remain active.
1161 If more time than this passes, then the container may end the conversation.
- 1162 • ***singlePrincipal (optional)*** - If true, only the principal (the user) that started the
1163 conversation has authority to continue the conversation. The default value is false.

1164

1165 The two attributes that take a time express the time as a string that starts with an integer, is
1166 followed by a space and then one of the following: "seconds", "minutes", "hours", "days" or
1167 "years".

1168

1169 Not specifying timeouts means that timeouts are defined by the SCA runtime implementation,
1170 however it chooses to do so.

1171

1172 The following snippet shows the use of the `@ConversationAttributes` annotation to set the
1173 maximum age for a Conversation to be 30 days.

1174

```

1175 package service.shoppingcart;
1176
1177 import org.esea.oasisopen.sca.annotations.ConversationAttributes;
1178
1179 @ConversationAttributes (maxAge="30 days");
1180 public class ShoppingCartServiceImpl implements ShoppingCartService {
1181     ...
1182 }

```

1183 8.8 @ConversationID

1184 The following Java code defines the `@ConversationID` annotation:

1185

```

1186 package org.esea.oasisopen.sca.annotations;
1187
1188 import static java.lang.annotation.ElementType.METHOD;
1189 import static java.lang.annotation.ElementType.FIELD;
1190 import static java.lang.annotation.RetentionPolicy.RUNTIME;

```

```

1191     import java.lang.annotation.Retention;
1192     import java.lang.annotation.Target;
1193
1194     @Target({METHOD, FIELD})
1195     @Retention(RUNTIME)
1196     public @interface ConversationID {
1197
1198     }
1199

```

1200 The @ConversationID annotation is used to annotate a Java class field or setter method that is
1201 used to inject the conversation ID. System generated conversation IDs are always strings, but
1202 application generated conversation IDs may be other complex types.

1203 The following snippet shows a conversation ID field definition sample.

```

1204
1205     @ConversationID
1206     private String conversationID;

```

1207
1208 The type of the field is not necessarily String.

1209

1210 8.9 @Destroy

1211 The following Java code defines the **@Destroy** annotation:

1212

```

1213     package org.oasisopen.sca.annotations;
1214
1215     import static java.lang.annotation.ElementType.METHOD;
1216     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1217     import java.lang.annotation.Retention;
1218     import java.lang.annotation.Target;
1219
1220     @Target (METHOD)
1221     @Retention (RUNTIME)
1222     public @interface Destroy {
1223
1224     }
1225

```

1226 The @Destroy annotation is used to denote a single Java class method that will be called when the
1227 scope defined for the implementation class ends. The method MAY have any access modifier and
1228 MUST have a void return type and no arguments.

1229 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1230 when the scope defined for the implementation class ends. If the implementation class has a
1231 method with an @Destroy annotation that does not match these criteria, the SCA runtime MUST
1232 NOT instantiate the implementation class.

1233

1234 The following snippet shows a sample for a destroy method definition.

1235

```

1236     @Destroy
1237     public void myDestroyMethod() {
1238         ...
1239     }

```

1240 8.10 @EagerInit

1241 The following Java code defines the **@EagerInit** annotation:

1242

```
1243 package org.esea.oasisopen.sca.annotations;
1244
1245 import static java.lang.annotation.ElementType.TYPE;
1246 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1247 import java.lang.annotation.Retention;
1248 import java.lang.annotation.Target;
1249
1250 @Target (TYPE)
1251 @Retention (RUNTIME)
1252 public @interface EagerInit {
1253
1254 }
```

1255 The **@EagerInit** annotation is used to annotate the Java class of a COMPOSITE scoped implementation for eager initialization. When marked for eager initialization, the composite scoped instance is created when its containing component is started.

1259 8.11 @EndsConversation

1260 The following Java code defines the **@EndsConversation** annotation:

1261

```
1262 package org.esea.oasisopen.sca.annotations;
1263
1264 import static java.lang.annotation.ElementType.METHOD;
1265 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1266 import java.lang.annotation.Retention;
1267 import java.lang.annotation.Target;
1268
1269 @Target (METHOD)
1270 @Retention (RUNTIME)
1271 public @interface EndsConversation {
1272
1273
1274 }
1275
```

1276 The **@EndsConversation** annotation is used to denote a method on a Java interface that is called to end a conversation.

1277 The **@EndsConversation** annotation has no attributes.

1278 The following snippet shows a sample using the **@EndsConversation** annotation.

```
1280 package services.shoppingbasket;
1281
1282 import org.esea.oasisopen.sca.annotations.EndsConversation;
1283
1284 public interface ShoppingBasket {
1285     void addItem(String itemID, int quantity);
1286
1287     @EndsConversation
1288     void buy();
1289 }
```


1290 8.12 @Init

1291 The following Java code defines the **@Init** annotation:

1292

```
1293 package org.esoaoasisopen.sca.annotations;
1294
1295 import static java.lang.annotation.ElementType.METHOD;
1296 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1297 import java.lang.annotation.Retention;
1298 import java.lang.annotation.Target;
1299
1300 @Target (METHOD)
1301 @Retention (RUNTIME)
1302 public @interface Init {
1303
1304
1305 }
1306
```

1307 The @Init annotation is used to denote a single Java class method that is called when the scope
1308 defined for the implementation class starts. The method MAY have any access modifier and MUST
1309 have a void return type and no arguments.

1310 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1311 after all property and reference injection is complete. If the implementation class has a method
1312 with an @Init annotation that does not match these criteria, the SCA runtime MUST NOT
1313 instantiate the implementation class.

1314 The following snippet shows an example of an init method definition.

1315

```
1316 @Init
1317 public void myInitMethod() {
1318     ...
1319 }
```

1320 8.13 @OneWay

1321 The following Java code defines the **@OneWay** annotation:

1322

```
1323 package org.esoaoasisopen.sca.annotations;
1324
1325 import static java.lang.annotation.ElementType.METHOD;
1326 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1327 import java.lang.annotation.Retention;
1328 import java.lang.annotation.Target;
1329
1330 @Target (METHOD)
1331 @Retention (RUNTIME)
1332 public @interface OneWay {
1333
1334
1335 }
1336
```

1337 The @OneWay annotation is used on a Java interface or class method to indicate that invocations
1338 will be dispatched in a non-blocking fashion as described in the section on Asynchronous
1339 Programming.

1340 The @OneWay annotation has no attributes.
1341 The following snippet shows the use of the @OneWay annotation on an interface.

```
1342 package services.hello;  
1343  
1344 import org.esea.oasisopen.sca.annotations.OneWay;  
1345  
1346 public interface HelloService {  
1347     @OneWay  
1348     void hello(String name);  
1349 }
```

1350 8.14 @Property

1351 The following Java code defines the @Property annotation:

```
1352 package org.esea.oasisopen.sca.annotations;  
1353  
1354 import static java.lang.annotation.ElementType.METHOD;  
1355 import static java.lang.annotation.ElementType.FIELD;  
1356 import static java.lang.annotation.ElementType.PARAMETER;  
1357 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1358 import java.lang.annotation.Retention;  
1359 import java.lang.annotation.Target;  
1360  
1361 @Target({METHOD, FIELD, PARAMETER})  
1362 @Retention(RUNTIME)  
1363 public @interface Property {  
1364  
1365     String name() default "";  
1366     boolean required() default true;  
1367 }  
1368  
1369  
1370
```

1371 The @Property annotation is used to denote a Java class field, a setter method, or a constructor
1372 parameter that is used to inject an SCA property value. The type of the property injected, which
1373 can be a simple Java type or a complex Java type, is defined by the type of the Java class field or
1374 the type of the input parameter of the setter method or constructor.

1375 The @Property annotation may be used on fields, on setter methods or on a constructor method
1376 parameter. However, the @Property annotation MUST NOT be used on a class field that is declared
1377 as final.

1378 Properties may also be injected via setter methods even when the @Property annotation is not
1379 present. However, the @Property annotation must be used in order to inject a property onto a
1380 non-public field. In the case where there is no @Property annotation, the name of the property is
1381 the same as the name of the field or setter.

1382 Where there is both a setter method and a field for a property, the setter method is used.

1383

1384 The @Property annotation has the following attributes:

- 1385 • **name (optional)** – the name of the property. For a field annotation, the default is the
1386 name of the field of the Java class. For a setter method annotation, the default is the
1387 JavaBeans property name corresponding to the setter method name. For a constructor
1388 parameter annotation, there is no default and the name attribute MUST be present.

- 1389 • **required (optional)** – specifies whether injection is required, defaults to true. For a
1390 constructor parameter annotation, this attribute MUST have the value true.

1391

1392 The following snippet shows a property field definition sample.

1393

```
1394 @Property(name="currency", required=true)  
1395 protected String currency;
```

1396

1397 The following snippet shows a property setter sample

1398

```
1399 @Property(name="currency", required=true)  
1400 public void setCurrency( String theCurrency ) {
```

```
1401     ....
```

```
1402 }
```

1403

1404 If the property is defined as an array or as any type that extends or implements
1405 **java.util.Collection**, then the implied component type has a property with a **many** attribute set to
1406 true.

1407

1408 The following snippet shows the definition of a configuration property using the @Property
1409 annotation for a collection.

1410

```
1411 ...  
1412 private List<String> helloConfigurationProperty;  
1413  
1414 @Property(required=true)  
1415 public void setHelloConfigurationProperty(List<String> property) {  
1416     helloConfigurationProperty = property;  
1417 }  
1418 ...
```

1419 8.15 @Reference

1420 The following Java code defines the **@Reference** annotation:

1421

```
1422 package org.esea.oasisopen.sca.annotations;  
1423  
1424 import static java.lang.annotation.ElementType.METHOD;  
1425 import static java.lang.annotation.ElementType.FIELD;  
1426 import static java.lang.annotation.ElementType.PARAMETER;  
1427 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1428 import java.lang.annotation.Retention;  
1429 import java.lang.annotation.Target;  
1430 @Target({METHOD, FIELD, PARAMETER})  
1431 @Retention(RUNTIME)  
1432 public @interface Reference {  
1433  
1434     String name() default "";  
1435     boolean required() default true;
```

1436 }
1437

1438 The @Reference annotation type is used to annotate a Java class field, a setter method, or a
1439 constructor parameter that is used to inject a service that resolves the reference. The interface of
1440 the service injected is defined by the type of the Java class field or the type of the input parameter
1441 of the setter method or constructor.

1442 The @Reference annotation MUST NOT be used on a class field that is declared as final.

1443 References may also be injected via setter methods even when the @Reference annotation is not
1444 present. However, the @Reference annotation must be used in order to inject a reference onto a
1445 non-public field. In the case where there is no @Reference annotation, the name of the reference
1446 is the same as the name of the field or setter.

1447 Where there is both a setter method and a field for a reference, the setter method is used.

1448 The @Reference annotation has the following attributes:

- 1449 • **name (optional)** – the name of the reference. For a field annotation, the default is the
1450 name of the field of the Java class. For a setter method annotation, the default is the
1451 JavaBeans property name corresponding to the setter method name. For a constructor
1452 parameter annotation, there is no default and the name attribute MUST be present.
- 1453 • **required (optional)** – whether injection of service or services is required. Defaults to true.
1454 For a constructor parameter annotation, this attribute MUST have the value true.

1455

1456 The following snippet shows a reference field definition sample.

1457

```
1458 @Reference(name="stockQuote", required=true)  
1459 protected StockQuoteService stockQuote;
```

1460

1461 The following snippet shows a reference setter sample

1462

```
1463 @Reference(name="stockQuote", required=true)  
1464 public void setStockQuote( StockQuoteService theSQService ) {  
1465     ...  
1466 }
```

1467

1468 The following fragment from a component implementation shows a sample of a service reference
1469 using the @Reference annotation. The name of the reference is "helloService" and its type is
1470 HelloService. The clientMethod() calls the "hello" operation of the service referenced by the
1471 helloService reference.

1472

```
1473 package services.hello;  
1474  
1475 private HelloService helloService;  
1476  
1477 @Reference(name="helloService", required=true)  
1478 public setHelloService(HelloService service) {  
1479     helloService = service;  
1480 }  
1481  
1482 public void clientMethod() {  
1483     String result = helloService.hello("Hello World!");
```

```
1484     ...
1485 }
1486
```

1487 The presence of a @Reference annotation is reflected in the componentType information that the
1488 runtime generates through reflection on the implementation class. The following snippet shows
1489 the component type for the above component implementation fragment.

```
1490
1491 <?xml version="1.0" encoding="ASCII"?>
1492 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1493     <!-- Any services offered by the component would be listed here -->
1494     <reference name="helloService" multiplicity="1..1">
1495         <interface.java interface="services.hello.HelloService"/>
1496     </reference>
1497
1498 </componentType>
1499
1500
```

1501 If the reference is not an array or collection, then the implied component type has a reference
1502 with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference **required**
1503 attribute – 1..1 applies if required=true.

1504
1505 If the reference is defined as an array or as any type that extends or implements **java.util.Collection**,
1506 then the implied component type has a **multiplicity** of either **1..n** or **0..n**, depending
1507 on whether the **required** attribute of the @Reference annotation is set to true or false – 1..n applies if
1508 required=true.

1509
1510 The following fragment from a component implementation shows a sample of a service reference
1511 definition using the @Reference annotation on a java.util.List. The name of the reference is
1512 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
1513 services referenced by the helloServices reference. In this case, at least one HelloService should
1514 be present, so **required** is true.

```
1515     @Reference(name="helloServices", required=true)
1516     protected List<HelloService> helloServices;
1517
1518     public void clientMethod() {
1519
1520         ...
1521         for (int index = 0; index < helloServices.size(); index++) {
1522             HelloService helloService =
1523                 (HelloService)helloServices.get(index);
1524             String result = helloService.hello("Hello World!");
1525         }
1526         ...
1527     }
1528
1529
```

1530 The following snippet shows the XML representation of the component type reflected from for the
1531 former component implementation fragment. There is no need to author this component type in
1532 this case since it can be reflected from the Java class.

```
1533
1534 <?xml version="1.0" encoding="ASCII"?>
1535 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1536
```

```
1537     <!-- Any services offered by the component would be listed here -->
1538     <reference name="helloServices" multiplicity="1..n">
1539         <interface.java interface="services.hello.HelloService"/>
1540     </reference>
1541
1542 </componentType>
```

1543 At runtime, the representation of an unwired reference depends on the reference's multiplicity. An
1544 unwired reference with a multiplicity of 0..1 must be null. An unwired reference with a multiplicity
1545 of 0..N must be an empty array or collection.
1546

1547 8.15.1 Reinjection

1548 References MAY be reinjected after the initial creation of a component if the reference target
1549 changes due to a change in wiring that has occurred since the component was initialized. In order
1550 for reinjection to occur, the following MUST be true:

- 1551 1. The component MUST NOT be STATELESS scoped.
- 1552 2. The reference MUST use either field-based injection or setter injection. References that are
1553 injected through constructor injection MUST NOT be changed. Setter injection allows for
1554 code in the setter method to perform processing in reaction to a change.
- 1555 3. If the reference has a conversational interface, then reinjection MUST NOT occur while the
1556 conversation is active.

1557 If a reference target changes and the reference is not reinjected, the reference MUST continue to
1558 work as if the reference target was not changed.

1559 If an operation is called on a reference where the target of that reference has been undeployed,
1560 the SCA runtime SHOULD throw `InvalidServiceException`. If an operation is called on a reference
1561 where the target of the reference has become unavailable for some reason, the SCA runtime
1562 SHOULD throw `ServiceUnavailableException`. If the target of the reference is changed, the
1563 reference MAY continue to work, depending on the runtime and the type of change that was made.
1564 If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.

1565 A `ServiceReference` that has been obtained from a reference by `ComponentContext.cast()`
1566 corresponds to the reference that is passed as a parameter to `cast()`. If the reference is
1567 subsequently reinjected, the `ServiceReference` obtained from the original reference MUST continue
1568 to work as if the reference target was not changed. If the target of a `ServiceReference` has been
1569 undeployed, the SCA runtime SHOULD throw `InvalidServiceException` when an operation is
1570 invoked on the `ServiceReference`. If the target of a `ServiceReference` has become unavailable, the
1571 SCA runtime SHOULD throw `ServiceUnavailableException` when an operation is invoked on the
1572 `ServiceReference`. If the target of a `ServiceReference` is changed, the reference MAY continue to
1573 work, depending on the runtime and the type of change that was made. If it doesn't work, the
1574 exception thrown will depend on the runtime and the cause of the failure.

1575 A reference or `ServiceReference` accessed through the component context by calling `getService()`
1576 or `getServiceReference()` MUST correspond to the current configuration of the domain. This
1577 applies whether or not reinjection has taken place. If the target has been undeployed or has
1578 become unavailable, the result SHOULD be a reference to the undeployed or unavailable service,
1579 and attempts to call business methods SHOULD throw an exception as described above. If the
1580 target has changed, the result SHOULD be a reference to the changed service.

1581 The rules for reference reinjection also apply to references with a multiplicity of 0..N or 1..N. This
1582 means that in the cases listed above where reference reinjection is not allowed, the array or
1583 `Collection` for the reference MUST NOT change its contents. In cases where the contents of a
1584 reference collection MAY change, then for references that use setter injection, the setter method
1585 MUST be called for any change to the contents. The reinjected array or `Collection` MUST NOT be
1586 the same array or `Collection` object previously injected to the component.

1587

Change event	Effect on		
	Reference	Existing ServiceReference Object	Subsequent invocations of ComponentContext.getServiceReference() or getService()
Change to the target of the reference	MAY be reinjected (if other conditions* apply). If not reinjected, then it MUST continue to work as if the reference target was not changed.	MUST continue to work as if the reference target was not changed.	Result corresponds to the current configuration of the domain.
Target service undeployed	Business methods SHOULD throw InvalidServiceException.	Business methods SHOULD throw InvalidServiceException.	Result SHOULD be a reference to the undeployed or unavailable service. Business methods SHOULD throw InvalidServiceException.
Target service changed	MAY continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.	MAY continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.	Result SHOULD be a reference to the changed service.
<p>* Other conditions:</p> <ol style="list-style-type: none"> 1. The component MUST NOT be STATELESS scoped. 2. The reference MUST use either field-based injection or setter injection. References that are injected through constructor injection MUST NOT be changed. <p>** Result of invoking ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().</p>			

1588

1589 8.16 @Remotable

1590 The following Java code defines the **@Remotable** annotation:

1591

1592 `package org.esoasisopen.sca.annotations;`

1593

1594 `import static java.lang.annotation.ElementType.TYPE;`

1595 `import static java.lang.annotation.RetentionPolicy.RUNTIME;`

1596 `import java.lang.annotation.Retention;`

1597 `import java.lang.annotation.Target;`

1598

1599

1600 `@Target (TYPE)`

1601 `@Retention (RUNTIME)`

1602 `public @interface Remotable {`

1603

1604 `}`

1605

1606 The @Remotable annotation is used to specify a Java service interface as remotable. A remotable
1607 service can be published externally as a service and must be translatable into a WSDL portType.

1608 The @Remotable annotation has no attributes.

1609

1610 The following snippet shows the Java interface for a remotable service with its @Remotable
1611 annotation.

```
1612 package services.hello;  
1613  
1614 import org.oesaoasisoasisopen.sca.annotations.*;  
1615  
1616 @Remotable  
1617 public interface HelloService {  
1618     String hello(String message);  
1619 }  
1620  
1621
```

1622 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
1623 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

1624

1625 Complex data types exchanged via remotable service interfaces MUST be compatible with the
1626 marshalling technology used by the service binding. For example, if the service is going to be
1627 exposed using the standard Web Service binding, then the parameters MAY be JAXB [JAX-B] types
1628 or Service Data Objects (SDOs) [SDO].

1629 Independent of whether the remotable service is called from outside of the composite that
1630 contains it or from another component in the same composite, the data exchange semantics are
1631 **by-value**.

1632 Implementations of remotable services may modify input data during or after an invocation and
1633 may modify return data after the invocation. If a remotable service is called locally or remotely,
1634 the SCA container is responsible for making sure that no modification of input data or post-
1635 invocation modifications to return data are seen by the caller.

1636

1637 The following snippet shows a remotable Java service interface.

1638

```
1639 package services.hello;  
1640  
1641 import org.oesaoasisoasisopen.sca.annotations.*;  
1642  
1643 @Remotable  
1644 public interface HelloService {  
1645     String hello(String message);  
1646 }  
1647  
1648 package services.hello;  
1649  
1650 import org.oesaoasisoasisopen.sca.annotations.*;  
1651  
1652 @Service(HelloService.class)  
1653 public class HelloServiceImpl implements HelloService {  
1654     public String hello(String message) {  
1655         ...  
1656     }  
1657
```


1658 }
1659 }

1660 8.17 @Scope

1661 The following Java code defines the **@Scope** annotation:

```
1662  
1663        package org.esoaoasisopen.sca.annotations;  
1664  
1665        import static java.lang.annotation.ElementType.TYPE;  
1666        import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1667        import java.lang.annotation.Retention;  
1668        import java.lang.annotation.Target;  
1669  
1670        @Target (TYPE)  
1671        @Retention (RUNTIME)  
1672        public @interface Scope {  
1673  
1674            String value() default "STATELESS";  
1675        }
```

1676 The @Scope annotation may only be used on a service's implementation class. It is an error to use
1677 this annotation on an interface.

1678 The @Scope annotation has the following attribute:

- 1679 • **value** – the name of the scope.
1680 For 'STATELESS' implementations, a different implementation instance may be used to
1681 service each request. Implementation instances may be newly created or be drawn from a
1682 pool of instances.
1683 SCA defines the following scope names, but others can be defined by particular Java-
1684 based implementation types:
1685 STATELESS
1686 COMPOSITE
1687 CONVERSATION

1688 The default value is STATELESS, except for an implementation offering a @Conversational service,
1689 which has a default scope of CONVERSATION. See section 2.2 for more details of the SCA-defined
1690 scopes.

1691 The following snippet shows a sample for a CONVERSATION scoped service implementation:

```
1692        package services.hello;  
1693  
1694        import org.esoaoasisopen.sca.annotations.*;  
1695  
1696        @Service (HelloService.class)  
1697        @Scope ("CONVERSATION")  
1698        public class HelloServiceImpl implements HelloService {  
1699  
1700            public String hello (String message) {  
1701                ...  
1702            }  
1703        }  
1704
```

1705 8.18 @Service

1706 The following Java code defines the **@Service** annotation:

1707

```

1708 package org.esea.oasisopen.sca.annotations;
1709
1710 import static java.lang.annotation.ElementType.TYPE;
1711 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1712 import java.lang.annotation.Retention;
1713 import java.lang.annotation.Target;
1714
1715 @Target (TYPE)
1716 @Retention (RUNTIME)
1717 public @interface Service {
1718
1719     Class<?>[] interfaces() default {};
1720     Class<?> value() default Void.class;
1721 }
1722

```

1723 The @Service annotation is used on a component implementation class to specify the SCA services
1724 offered by the implementation. The class need not be declared as implementing all of the
1725 interfaces implied by the services, but all methods of the service interfaces must be present. A
1726 class used as the implementation of a service is not required to have a @Service annotation. If a
1727 class has no @Service annotation, then the rules determining which services are offered and what
1728 interfaces those services have are determined by the specific implementation type.

1729 The @Service annotation has the following attributes:

- 1730 • **interfaces** – The value is an array of interface or class objects that should be exposed as
1731 services by this component.
- 1732 • **value** – A shortcut for the case when the class provides only a single service interface.

1733 Only one of these attributes should be specified.

1734

1735 A @Service annotation with no attributes is meaningless, it is the same as not having the
1736 annotation there at all.

1737 The **service names** of the defined services default to the names of the interfaces or class, without
1738 the package name.

1739 A component MUST NOT have two services with the same Java simple name. If a Java
1740 implementation needs to realize two services with the same Java simple name then this can be
1741 achieved through subclassing of the interface.

1742 The following snippet shows an implementation of the HelloService marked with the @Service
1743 annotation.

```

1744 package services.hello;
1745
1746 import org.esea.oasisopen.sca.annotations.Service;
1747
1748 @Service (HelloService.class)
1749 public class HelloServiceImpl implements HelloService {
1750
1751     public void hello (String name) {
1752         System.out.println ("Hello " + name);
1753     }
1754 }
1755

```

9 WSDL to Java and Java to WSDL

1756

1757 The SCA Client and Implementation Model for Java applies the WSDL to Java and Java to WSDL
1758 mapping rules as defined by the JAX-WS specification [JAX-WS] for generating remotable Java
1759 interfaces from WSDL portTypes and vice versa.

1760 For the purposes of the Java-to-WSDL mapping algorithm, the interface is treated as if it had a
1761 @WebService annotation on the class, even if it doesn't, and the
1762 @org.oasisopen.annotations.OneWay annotation should be treated as a synonym for the
1763 @javax.jws.OneWay annotation. For the WSDL-to-Java mapping, the generated @WebService
1764 annotation implies that the interface is @Remotable.

1765 For the mapping from Java types to XML schema types, SCA permits both the JAXB 2.1 [JAX-B]
1766 mapping and the SDO 2.1 [SDO] mapping. SCA runtimes MUST support the JAXB 2.1 mapping
1767 and MAY support the SDO 2.1 mapping. Having a choice of binding technologies is allowed, as
1768 noted in the first paragraph of section 5 of the JSR 181 (version 2) specification, which is
1769 referenced by the JAX-WS specification.

1770 The JAX-WS mappings are applied with the following restrictions:

- 1771 • No support for holders

1772

1773 **Note:** This specification needs more examples and discussion of how JAX-WS's client asynchronous
1774 model is used.

9.1 JAX-WS Client Asynchronous API for a Synchronous Service

1775
1776 The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client
1777 application with a means of invoking that service asynchronously, so that the client can invoke a service
1778 operation and proceed to do other work without waiting for the service operation to complete its
1779 processing. The client application can retrieve the results of the service either through a polling
1780 mechanism or via a callback method which is invoked when the operation completes.

1781 For SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional
1782 client-side asynchronous polling and callback methods defined by JAX-WS. For SCA service interfaces
1783 defined using interface.java, the Java interface MUST NOT contain these methods. If these methods are
1784 present, SCA Runtimes MUST NOT include them in the SCA reference interface as defined by the
1785 Assembly specification. These methods are recognized as follows.

1786 For each method M in the interface, if another method P in the interface has

- 1787 a. a method name that is M's method name with the characters "Async" appended, and
- 1788 b. the same parameter signature as M, and
- 1789 c. a return type of Response<R> where R is the return type of M

1790 then P is a JAX-WS polling method that isn't part of the SCA interface contract.

1791 For each method M in the interface, if another method C in the interface has

- 1792 a. a method name that is M's method name with the characters "Async" appended, and
- 1793 b. a parameter signature that is M's parameter signature with an additional final parameter of type
1794 AsyncHandler<R> where R is the return type of M, and
- 1795 c. a return type of Future<?>

1796 then C is a JAX-WS callback method that isn't part of the SCA interface contract.

1797 As an example, an interface may be defined in WSDL as follows:

```
1798 <!-- WSDL extract -->  
1799 <message name="getPrice">
```

```
1800 <part name="ticker" type="xsd:string"/>
1801 </message>
1802
1803 <message name="getPriceResponse">
1804 <part name="price" type="xsd:float"/>
1805 </message>
1806
1807 <portType name="StockQuote">
1808 <operation name="getPrice">
1809 <input message="tns:getPrice"/>
1810 <output message="tns:getPriceResponse"/>
1811 </operation>
1812 </portType>
```

1813

1814 The JAX-WS asynchronous mapping will produce the following Java interface:

```
1815 // asynchronous mapping
1816 @WebService
1817 public interface StockQuote {
1818     float getPrice(String ticker);
1819     Response<Float> getPriceAsync(String ticker);
1820     Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
1821 }
```

1822

1823 For SCA interface definition purposes, this is treated as equivalent to the following:

```
1824 // synchronous mapping
1825 @WebService
1826 public interface StockQuote {
1827     float getPrice(String ticker);
1828 }
```

1829

1830 SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above
1831 example, if the client implementation uses the asynchronous form of the interface, the two
1832 additional getPriceAsync() methods can be used for polling and callbacks as defined by the JAX-
1833 WS specification.

1834 10 Policy Annotations for Java

1835 SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which
1836 influence how implementations, services and references behave at runtime. The policy facilities
1837 are described in [the SCA Policy Framework specification \[POLICY\]](#). In particular, the facilities
1838 include Intents and Policy Sets, where intents express abstract, high-level policy requirements and
1839 policy sets express low-level detailed concrete policies.

1840 Policy metadata can be added to SCA assemblies through the means of declarative statements
1841 placed into Composite documents and into Component Type documents. These annotations are
1842 completely independent of implementation code, allowing policy to be applied during the assembly
1843 and deployment phases of application development.

1844 However, it can be useful and more natural to attach policy metadata directly to the code of
1845 implementations. This is particularly important where the policies concerned are relied on by the
1846 code itself. An example of this from the Security domain is where the implementation code
1847 expects to run under a specific security Role and where any service operations invoked on the
1848 implementation must be authorized to ensure that the client has the correct rights to use the
1849 operations concerned. By annotating the code with appropriate policy metadata, the developer
1850 can rest assured that this metadata is not lost or forgotten during the assembly and deployment
1851 phases.

1852 The SCA Java Common Annotations specification provides a series of annotations which provide
1853 the capability for the developer to attach policy information to Java implementation code. The
1854 annotations concerned first provide general facilities for attaching SCA Intents and Policy Sets to
1855 Java code. Secondly, there are further specific annotations that deal with particular policy intents
1856 for certain policy domains such as Security.

1857 The SCA Java Common Annotations specification supports using [the Common Annotation for Java
1858 Platform specification \(JSR-250\) \[JSR-250\]](#). An implication of adopting the common annotation
1859 for Java platform specification is that the SCA Java specification support consistent annotation and
1860 Java class inheritance relationships.

1861

1862 10.1 General Intent Annotations

1863 SCA provides the annotation **@Requires** for the attachment of any intent to a Java class, to a
1864 Java interface or to elements within classes and interfaces such as methods and fields.

1865 The @Requires annotation can attach one or multiple intents in a single statement.

1866 Each intent is expressed as a string. Intents are XML QNames, which consist of a Namespace URI
1867 followed by the name of the Intent. The precise form used follows the string representation used
1868 by the `javax.xml.namespace.QName` class, which is as follows:

1869 `"{" + Namespace URI + "}" + intentname`

1870 Intents may be qualified, in which case the string consists of the base intent name, followed by a
1871 ".", followed by the name of the qualifier. There may also be multiple levels of qualification.

1872 This representation is quite verbose, so we expect that reusable String constants will be defined
1873 for the namespace part of this string, as well as for each intent that is used by Java code. SCA
1874 defines constants for intents such as the following:

```
1875 public static final String SCA_PREFIX="{http://docs.oasis-  
1876 open.org/ns/opencsa/sca/200712}";
```

```
1877 public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
```

```
1878 public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
```

1879 Notice that, by convention, qualified intents include the qualifier as part of the name of the
1880 constant, separated by an underscore. These intent constants are defined in the file that defines
1881 an annotation for the intent (annotations for intents, and the formal definition of these constants,
1882 are covered in a following section).

1883 Multiple intents (qualified or not) are expressed as separate strings within an array declaration.

1884 An example of the @Requires annotation with 2 qualified intents (from the Security domain)
1885 follows:

```
1886  
1887     @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

1888
1889 This attaches the intents "confidentiality.message" and "integrity.message".
1890 The following is an example of a reference requiring support for confidentiality:

```
1891     package org.oesa.oasisopen.sca.annotation;  
1892  
1893     import static org.oesa.oasisopen.sca.annotation.Confidentiality.*;  
1894  
1895     public class Foo {  
1896         @Requires(CONFIDENTIALITY)  
1897         @Reference  
1898         public void setBar(Bar bar) {  
1899             ...  
1900         }  
1901     }
```

1902 Users may also choose to only use constants for the namespace part of the QName, so that they
1903 may add new intents without having to define new constants. In that case, this definition would
1904 instead look like this:

```
1905     package org.oesa.oasisopen.sca.annotation;  
1906  
1907     import static org.oesa.oasisopen.sca.Constants.*;  
1908  
1909     public class Foo {  
1910         @Requires(SCA_PREFIX+"confidentiality")  
1911         @Reference  
1912         public void setBar(Bar bar) {  
1913             ...  
1914         }  
1915     }
```

1916
1917 The formal syntax for the @Requires annotation follows:

```
1918     @Requires( "qualifiedIntent" | {"qualifiedIntent" [, "qualifiedIntent"]}
```

1919 where

1920 qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2

1921

1922 The following shows the formal definition of the @Requires annotation:

1923

```
1924 package org.esooasisopen.sca.annotation;
1925 import static java.lang.annotation.ElementType.TYPE;
1926 import static java.lang.annotation.ElementType.METHOD;
1927 import static java.lang.annotation.ElementType.FIELD;
1928 import static java.lang.annotation.ElementType.PARAMETER;
1929 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1930 import java.lang.annotation.Retention;
1931 import java.lang.annotation.Target;
1932 import java.lang.annotation.Inherited;
```

1933

```
1934 @Inherited
1935 @Retention(RUNTIME)
1936 @Target({TYPE, METHOD, FIELD, PARAMETER})
```

1937

```
1938 public @interface Requires {
1939     String[] value() default "";
1940 }
```

1941 The SCA_NS constant is defined in the Constants interface:

```
1942 package org.esooasisopen.sca;
1943
1944 public interface Constants {
1945     String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";
1946     String SCA_PREFIX = "{"+SCA_NS+"}";
1947 }
```

1948

1949 10.2 Specific Intent Annotations

1950 In addition to the general intent annotation supplied by the @Requires annotation described
1951 above, it is also possible to have Java annotations that correspond to specific policy intents. SCA
1952 provides a number of these specific intent annotations and it is also possible to create new specific
1953 intent annotations for any intent.

1954 The general form of these specific intent annotations is an annotation with a name derived from
1955 the name of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an
1956 attribute to the annotation in the form of a string or an array of strings.

1957 For example, the SCA confidentiality intent described in [the section on General Intent Annotations](#)
1958 using the @Requires(CONFIDENTIALITY) intent can also be specified with the specific
1959 @Confidentiality intent annotation. The specific intent annotation for the "integrity" security intent
1960 is:

1961 @Integrity

1962 An example of a qualified specific intent for the "authentication" intent is:

1963 @Authentication({"message", "transport"})

1964 This annotation attaches the pair of qualified intents: "authentication.message" and
 1965 "authentication.transport" (the sca: namespace is assumed in this both of these cases –
 1966 "http://docs.oasis-open.org/ns/opencsa/sca/200712").

1967 The general form of specific intent annotations is:

1968 @<Intent>[(qualifiers)]

1969 where Intent is an NCName that denotes a particular type of intent.

1970 Intent ::= NCName

1971 qualifiers ::= "qualifier" | {"qualifier" [, "qualifier"] }

1972 qualifier ::= NCName | NCName/qualifier

1973

10.2.1 How to Create Specific Intent Annotations

1974

1975 SCA identifies annotations that correspond to intents by providing an @Intent annotation which
 1976 must be used in the definition of an intent annotation.

1977 The @Intent annotation takes a single parameter, which (like the @Requires annotation) is the
 1978 String form of the QName of the intent. As part of the intent definition, it is good practice
 1979 (although not required) to also create String constants for the Namespace, the Intent and for
 1980 Qualified versions of the Intent (if defined). These String constants are then available for use with
 1981 the @Requires annotation and it should also be possible to use one or more of them as
 1982 parameters to the @Intent annotation.

1983 Alternatively, the QName of the intent may be specified using separate parameters for the
 1984 targetNamespace and the localPart for example:

```
@Intent(targetNamespace=SCA_NS, localPart="confidentiality").
```

1985

1986 The definition of the @Intent annotation is the following:

```
1987
1988        package org.oasisopen.sca.annotation;
1989        import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
1990        import static java.lang.annotation.RetentionPolicy.RUNTIME;
1991        import java.lang.annotation.Retention;
1992        import java.lang.annotation.Target;
1993        import java.lang.annotation.Inherited;
1994
1995        @Retention(RUNTIME)
1996        @Target(ANNOTATION_TYPE)
1997        public @interface Intent {
1998            String value() default "";
1999            String targetNamespace() default "";
2000            String localPart() default "";
2001        }
```


2002 When an intent can be qualified, it is good practice for the first attribute of the annotation to be a
2003 string (or an array of strings) which holds one or more qualifiers.

2004 In this case, the attribute's definition should be marked with the @Qualifier annotation. The
2005 @Qualifier tells SCA that the value of the attribute should be treated as a qualifier for the intent
2006 represented by the whole annotation. If more than one qualifier value is specified in an
2007 annotation, it means that multiple qualified forms are required. For example:

```
2008 @Confidentiality({"message", "transport"})
```

2009 implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport"
2010 are set for the element to which the confidentiality intent is attached.

2011 The following is the definition of the @Qualifier annotation.

2012

```
2013 package org.oasisoasisopen.sca.annotation;  
2014 import static java.lang.annotation.ElementType.METHOD;  
2015 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2016 import java.lang.annotation.Retention;  
2017 import java.lang.annotation.Target;  
2018 import java.lang.annotation.Inherited;
```

2019

```
2020 @Retention(RetentionPolicy.RUNTIME)
```

```
2021 @Target(ElementType.METHOD)
```

```
2022 public @interface Qualifier {
```

```
2023 }
```

2024

2025 Examples of the use of the @Intent and the @Qualifier annotations in the definition of specific
2026 intent annotations are shown in [the section dealing with Security Interaction Policy](#).

2027

2028 10.3 Application of Intent Annotations

2029 The SCA Intent annotations can be applied to the following Java elements:

- 2030 • Java class
- 2031 • Java interface
- 2032 • Method
- 2033 • Field

2034 Where multiple intent annotations (general or specific) are applied to the same Java element, they
2035 are additive in effect. An example of multiple policy annotations being used together follows:

```
2036 @Authentication
```

```
2037 @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

2038 In this case, the effective intents are "authentication", "confidentiality.message" and
2039 "integrity.message".

2040 If an annotation is specified at both the class/interface level and the method or field level, then
2041 the method or field level annotation completely overrides the class level annotation of the same
2042 type.

2043 The intent annotation can be applied either to classes or to class methods when adding annotated
2044 policy on SCA services. Applying an intent to the setter method in a reference injection approach
2045 allows intents to be defined at references.

2046 10.3.1 Inheritance And Annotation

2047 The inheritance rules for annotations are consistent with the common annotation specification, JSR
2048 250.

2049 The following example shows the inheritance relations of intents on classes, operations, and super
2050 classes.

```
2051
2052 package services.hello;
2053 import org.esea.oasisopen.sca.annotations.Remotable;
2054 import org.esea.oasisopen.sca.annotations.Integrity;
2055 import org.esea.oasisopen.sca.annotations.Authentication;
2056
2057 @Integrity("transport")
2058 @Authentication
2059 public class HelloService {
2060     @Integrity
2061     @Authentication("message")
2062     public String hello(String message) {...}
2063
2064     @Integrity
2065     @Authentication("transport")
2066     public String helloThere() {...}
2067 }
2068
2069 package services.hello;
2070 import org.esea.oasisopen.sca.annotations.Remotable;
2071 import org.esea.oasisopen.sca.annotations.Confidentiality;
2072 import org.esea.oasisopen.sca.annotations.Authentication;
2073
2074 @Confidentiality("message")
2075 public class HelloChildService extends HelloService {
2076     @Confidentiality("transport")
2077     public String hello(String message) {...}
2078     @Authentication
2079     String helloWorld() {...}
2080 }
```

2081 Example 2a. Usage example of annotated policy and inheritance.

2082
2083 The effective intent annotation on the helloWorld method is Integrity("transport"),
2084 @Authentication, and @Confidentiality("message").

2085 The effective intent annotation on the hello method of the HelloChildService is
2086 @Integrity("transport"), @Authentication, and @Confidentiality("transport"),
2087 The effective intent annotation on the helloThere method of the HelloChildService is @Integrity
2088 and @Authentication("transport"), the same as in HelloService class.
2089 The effective intent annotation on the hello method of the HelloService is @Integrity and
2090 @Authentication("message")

2091
2092 The listing below contains the equivalent declarative security interaction policy of the HelloService
2093 and HelloChildService implementation corresponding to the Java interfaces and classes shown in
2094 Example 2a.

```
2095  
2096 <?xml version="1.0" encoding="ASCII"?>  
2097  
2098 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2099           name="HelloServiceComposite" >  
2100   <service name="HelloService" requires="integrity/transport  
2101           authentication">  
2102     ...  
2103   </service>  
2104   <service name="HelloChildService" requires="integrity/transport  
2105           authentication confidentiality/message">  
2106     ...  
2107   </service>  
2108   ...  
2109  
2110   <component name="HelloServiceComponent">*  
2111     <implementation.java class="services.hello.HelloService"/>  
2112       <operation name="hello" requires="integrity  
2113           authentication/message"/>  
2114       <operation name="helloThere"  
2115 requires="integrity  
2116           authentication/transport"/>  
2117     </component>  
2118   <component name="HelloChildServiceComponent">*  
2119     <implementation.java  
2120 class="services.hello.HelloChildService" />  
2121     <operation name="hello"  
2122 requires="confidentiality/transport"/>  
2123     <operation name="helloThere" requires=" integrity/transport  
2124           authentication"/>  
2125     <operation name="helloWorld" requires="authentication"/>  
2126   </component>  
2127   ...  
2128   ...  
2129 </composite>
```

2130
2131 Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.
2132
2133

2134 10.4 Relationship of Declarative And Annotated Intents

2135 Annotated intents on a Java class cannot be overridden by declarative intents either in a
2136 composite document which uses the class as an implementation or by statements in a component

2137 Type document associated with the class. This rule follows the general rule for intents that they
2138 represent fundamental requirements of an implementation.

2139 An unqualified version of an intent expressed through an annotation in the Java class may be
2140 qualified by a declarative intent in a using composite document.

2141

2142 10.5 Policy Set Annotations

2143 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies (for
2144 example, a concrete policy is the specific encryption algorithm to use when encrypting messages
2145 when using a specific communication protocol to link a reference to a service).

2146
2147 Policy Sets can be applied directly to Java implementations using the **@PolicySets** annotation.
2148 The @PolicySets annotation either takes the QName of a single policy set as a string or the name
2149 of two or more policy sets as an array of strings:

```
2150     @PolicySets( "<policy set QName>" |  
2151                 { "<policy set QName>" [, "<policy set QName>"] })
```

2152

2153 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

2154 An example of the @PolicySets annotation:

2155

```
2156     @Reference(name="helloService", required=true)  
2157     @PolicySets({ MY_NS + "WS_Encryption_Policy",  
2158                 MY_NS + "WS_Authentication_Policy" })  
2159     public setHelloService(HelloService service) {  
2160         . . .  
2161     }
```

2162 In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both
2163 using the namespace defined for the constant MY_NS.

2164 PolicySets must satisfy intents expressed for the implementation when both are present, according
2165 to the rules defined in [the Policy Framework specification \[POLICY\]](#).

2166 The SCA Policy Set annotation can be applied to the following Java elements:

- 2167 • Java class
- 2168 • Java interface
- 2169 • Method
- 2170 • Field

2171

2172 10.6 Security Policy Annotations

2173 This section introduces annotations for SCA's security intents, as defined in [the SCA Policy
2174 Framework specification \[POLICY\]](#).

2175

2176 10.6.1 Security Interaction Policy

2177 The following interaction policy Intents and qualifiers are defined for Security Policy, which apply
2178 to the operation of services and references of an implementation:

- 2179 • @Integrity
- 2180 • @Confidentiality
- 2181 • @Authentication

2182 All three of these intents have the same pair of Qualifiers:

- 2183 • message
- 2184 • transport

2185 The following snippets shows the @Integrity, @Confidentiality and @Authentication annotations:

```

2186 package org. esoaoasisopen.sca.annotation;
2187
2188 import java.lang.annotation.*;
2189 import static org. esoaoasisopen.sca.Constants.SCA_NS;
2190
2191 @Inherited
2192 @Retention(RetentionPolicy.RUNTIME)
2193 @Target({ElementType.TYPE, ElementType.METHOD,
2194          ElementType.FIELD, ElementType.PARAMETER})
2195 @Intent(Integrity.INTEGRITY)
2196 public @interface Integrity {
2197     String INTEGRITY = SCA_NS+"integrity";
2198     String INTEGRITY_MESSAGE = INTEGRITY+".message";
2199     String INTEGRITY_TRANSPORT = INTEGRITY+".transport";
2200     @Qualifier
2201     String[] value() default "";
2202 }
2203
2204
2205 package org. esoaoasisopen.sca.annotation;
2206
2207 import java.lang.annotation.*;
2208 import static org. esoaoasisopen.sca.Constants.SCA_NS;
2209
2210 @Inherited
2211 @Retention(RetentionPolicy.RUNTIME)
2212 @Target({ElementType.TYPE, ElementType.METHOD,
2213          ElementType.FIELD, ElementType.PARAMETER})
2214 @Intent(Confidentiality.CONFIDENTIALITY)
2215 public @interface Confidentiality {
2216     String CONFIDENTIALITY = SCA_NS+"confidentiality";
2217     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY+".message";

```

```

2218     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY+".transport";
2219     @Qualifier
2220     String[] value() default "";
2221 }
2222
2223
2224 package org.escaoasisopen.sca.annotation;
2225
2226 import java.lang.annotation.*;
2227 import static org.escaoasisopen.sca.Constants.SCA_NS;
2228
2229 @Inherited
2230 @Retention(RetentionPolicy.RUNTIME)
2231 @Target({ElementType.TYPE, ElementType.METHOD,
2232          ElementType.FIELD, ElementType.PARAMETER})
2233 @Intent(Authentication.AUTHENTICATION)
2234 public @interface Authentication {
2235     String AUTHENTICATION = SCA_NS+"authentication";
2236     String AUTHENTICATION_MESSAGE = AUTHENTICATION+".message";
2237     String AUTHENTICATION_TRANSPORT = AUTHENTICATION+".transport";
2238     @Qualifier
2239     String[] value() default "";
2240 }

```

2241
2242

2243 The following example shows an example of applying an intent to the setter method used to inject
2244 a reference. Accessing the hello operation of the referenced HelloService requires both
2245 "integrity.message" and "authentication.message" intents to be honored.

```

2246
2247 //Interface for HelloService
2248 public interface service.hello.HelloService {
2249     String hello(String helloMsg);
2250 }
2251
2252 // Interface for ClientService
2253 public interface service.client.ClientService {
2254     public void clientMethod();
2255 }
2256
2257 // Implementation class for ClientService
2258 package services.client;

```

```

2259
2260     import services.hello.HelloService;
2261
2262     import org.oasisoasisopen.sca.annotations.*;
2263
2264     @Service(ClientService.class)
2265     public class ClientServiceImpl implements ClientService {
2266
2267
2268         private HelloService helloService;
2269
2270         @Reference(name="helloService", required=true)
2271         @Integrity("message")
2272         @Authentication("message")
2273         public void setHelloService(HelloService service) {
2274             helloService = service;
2275         }
2276
2277         public void clientMethod() {
2278             String result = helloService.hello("Hello World!");
2279             ...
2280         }
2281     }

```

2282
2283 Example 1. Usage of annotated intents on a reference.
2284

2285 10.6.2 Security Implementation Policy

2286 SCA defines a number of security policy annotations that apply as policies to implementations
2287 themselves. These annotations mostly have to do with authorization and security identity. The
2288 following authorization and security identity annotations (as defined in JSR 250) are supported:

- 2289 • RunAs

2290 Takes as a parameter a string which is the name of a Security role.
2291 eg. @RunAs("Manager")
2292

- 2293 • Code marked with this annotation will execute with the Security permissions of the
2294 identified role.

- 2295 • RolesAllowed

2296 Takes as a parameter a single string or an array of strings which represent one or more
2297 role names. When present, the implementation can only be accessed by principals whose
2298 role corresponds to one of the role names listed in the @roles attribute. How role names
2299 are mapped to security principals is implementation dependent (SCA does not define this).
2300 eg. @RolesAllowed({"Manager", "Employee"})
2301

- 2302 • PermitAll

2303 No parameters. When present, grants access to all roles.
2304

- 2305 • DenyAll
- 2306
- 2307 No parameters. When present, denies access to all roles.
- 2308 • DeclareRoles
- 2309 Takes as a parameter a string or an array of strings which identify one or more role names
- 2310 that form the set of roles used by the implementation.
- 2311 eg. @DeclareRoles({"Manager", "Employee", "Customer"})
- 2312 (all these are declared in the Java package javax.annotation.security)
- 2313 For a full explanation of these intents, see [the Policy Framework specification \[POLICY\]](#).

2314 10.6.2.1 Annotated Implementation Policy Example

2315 The following is an example showing annotated security implementation policy:

```
2316
2317 package services.account;
2318 @Remotable
2319 public interface AccountService {
2320     AccountReport getAccountReport(String customerID);
2321 }
```

2322

2323 The following is a full listing of the AccountServiceImpl class, showing the Service it implements,

2324 plus the service references it makes and the settable properties that it has, along with a set of

2325 implementation policy annotations:

```
2326
2327 package services.account;
2328 import java.util.List;
2329 import commonj.sdo.DataFactory;
2330 import org.esea.oasisopen.sca.annotations.Property;
2331 import org.esea.oasisopen.sca.annotations.Reference;
2332 import org.esea.oasisopen.sca.annotations.RolesAllowed;
2333 import org.esea.oasisopen.sca.annotations.RunAs;
2334 import org.esea.oasisopen.sca.annotations.PermitAll;
2335 import services.accountdata.AccountDataService;
2336 import services.accountdata.CheckingAccount;
2337 import services.accountdata.SavingsAccount;
2338 import services.accountdata.StockAccount;
2339 import services.stockquote.StockQuoteService;
2340 @RolesAllowed("customers")
2341 @RunAs("accountants" )
2342 public class AccountServiceImpl implements AccountService {
2343
2344     @Property
2345     protected String currency = "USD";
2346
2347     @Reference
2348     protected AccountDataService accountDataService;
```



```

2349     @Reference
2350     protected StockQuoteService stockQuoteService;
2351
2352     @RolesAllowed({"customers", "accountants"})
2353     public AccountReport getAccountReport(String customerID) {
2354
2355         DataFactory dataFactory = DataFactory.INSTANCE;
2356         AccountReport accountReport =
2357             (AccountReport) dataFactory.create (AccountReport.class);
2358         List accountSummaries = accountReport.getAccountSummaries();
2359
2360         CheckingAccount checkingAccount =
2361             accountDataService.getCheckingAccount (customerID);
2362         AccountSummary checkingAccountSummary =
2363             (AccountSummary) dataFactory.create (AccountSummary.class);
2364         checkingAccountSummary.setAccountNumber (checkingAccount.getAccountNumber ()
2365 );
2366
2367         checkingAccountSummary.setAccountType ("checking");
2368         checkingAccountSummary.setBalance (fromUSDollarToCurrency
2369             (checkingAccount.getBalance ()));
2370         accountSummaries.add (checkingAccountSummary);
2371
2372         SavingsAccount savingsAccount =
2373             accountDataService.getSavingsAccount (customerID);
2374         AccountSummary savingsAccountSummary =
2375             (AccountSummary) dataFactory.create (AccountSummary.class);
2376
2377         savingsAccountSummary.setAccountNumber (savingsAccount.getAccountNumber ());
2378         savingsAccountSummary.setAccountType ("savings");
2379         savingsAccountSummary.setBalance (fromUSDollarToCurrency
2380             (savingsAccount.getBalance ()));
2381         accountSummaries.add (savingsAccountSummary);
2382
2383         StockAccount stockAccount =
2384         accountDataService.getStockAccount (customerID);
2385         AccountSummary stockAccountSummary =
2386             (AccountSummary) dataFactory.create (AccountSummary.class);
2387         stockAccountSummary.setAccountNumber (stockAccount.getAccountNumber ());
2388         stockAccountSummary.setAccountType ("stock");
2389         float balance= (stockQuoteService.getQuote (stockAccount.getSymbol ())) *
2390             stockAccount.getQuantity ();
2391         stockAccountSummary.setBalance (fromUSDollarToCurrency (balance) );
2392         accountSummaries.add (stockAccountSummary);

```

```
2393
2394     return accountReport;
2395 }
2396
2397 @PermitAll
2398 public float fromUSDollarToCurrency(float value) {
2399
2400     if (currency.equals("USD")) return value; else
2401     if (currency.equals("EURO")) return value * 0.8f; else
2402     return 0.0f;
2403 }
2404 }
```

2405 Example 3. Usage of annotated security implementation policy for the java language.

2406 In this example, the implementation class as a whole is marked:

- 2407 • @RolesAllowed("customers") - indicating that customers have access to the
- 2408 implementation as a whole
- 2409 • @RunAs("accountants") - indicating that the code in the implementation runs with the
- 2410 permissions of accountants

2411 The getAccountReport(..) method is marked with @RolesAllowed({"customers", "accountants"}),

2412 which indicates that this method can be called by both customers and accountants.

2413 The fromUSDollarToCurrency() method is marked with @PermitAll, which means that this method

2414 can be called by any role.

2415

2416

A. XML Schema: sca-interface-java.xsd

```
2417 <?xml version="1.0" encoding="UTF-8"?>
2418 <!-- (c) Copyright SCA Collaboration 2006 -->
2419 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2420     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2421     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2422     elementFormDefault="qualified">
2423
2424     <include schemaLocation="sca-core.xsd"/>
2425
2426     <element name="interface.java" type="sca:JavaInterface"
2427         substitutionGroup="sca:interface"/>
2428     <complexType name="JavaInterface">
2429         <complexContent>
2430             <extension base="sca:Interface">
2431                 <sequence>
2432                     <any namespace="##other" processContents="lax"
2433 minOccurs="0" maxOccurs="unbounded"/>
2434                 </sequence>
2435                 <attribute name="interface" type="NCName" use="required"/>
2436                 <attribute name="callbackInterface" type="NCName"
2437 use="optional"/>
2438                 <anyAttribute namespace="##any" processContents="lax"/>
2439             </extension>
2440         </complexContent>
2441     </complexType>
2442 </schema>
2443
```

2444

B. Acknowledgements

2445 The following individuals have participated in the creation of this specification and are gratefully
2446 acknowledged:

2447 **Participants:**

2448 [Participant Name, Affiliation | Individual Member]

2449 [Participant Name, Affiliation | Individual Member]

2450

C. Non-Normative Text

2452

D. Revision History

2453 [optional; should not be included in OASIS Standards]

2454

Revision	Date	Editor	Changes Made
1	2007-09-26	Anish Karmarkar	Applied the OASIS template + related changes to the Submission
2	2008-02-28	Anish Karmarkar	Applied resolution of issues: 4, 11, and 26
3	2008-04-17	Mike Edwards	Ed changes
4	2008-05-27	Anish Karmarkar David Booz Mark Combella	Added InvalidServiceException in Section 7 Various editorial updates
WD04	2008-08-15	Anish Karmarkar	* Applied resolution of issue 9 (it was applied before, not sure by whom, but it was applied incorrectly) * Applied resolution of issue 12, 22, 23, 29, 31, 35, 36, 37, 44, 45 * Note that issue 33 was applied, but not noted, in a previous version * Replaced the osoa.org NS with the oasis-open.org NS
WD05	2008-10-03	Anish Karmarkar	* Fixed the resolution of issue 37 but re-adding the sentence: "However, the @... annotation must be used in order to inject a property onto a non-public field. -- in the @Property and @Reference section * resolution of issue 9 was applied incorrectly. Fixed that -- removed the requirement for throwing an exception on ComponentContext.getServiceReferences() when multiplicity of references > 1 * minor ed changes
cd01-rev1	2008-12-11	Anish Karmarkar	* Fixed reference style to [RFC2119] instead of [1]. * Applied resolutions of issues 20, 21, 41, 42, 43, 47, 48, 49.
cd01-rev2	2008-12-12	Anish Karmarkar	* Applied resolutions of issues 61, 71, 72, 73, 79, 81, 82, 84, 112
cd01-rev3	2008-12-16	David Booz	* Applied resolution of issues 56, 75, 111
<u>cd01-rev4</u>	<u>2009-01-18</u>	<u>Anish Karmarkar</u>	<u>* Applied resolutions of issues 28, 52, 94, 96, 99, 101</u>

2455

2456