



Service Component Architecture SCA-J Common Annotations and APIs Specification Version 1.1

Committee Draft 03 – Rev3

01 February 2010

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd03.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd03.doc>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd03.pdf> (Authoritative)

Previous Version:

<http://www.oasis-open.org/apps/org/workgroup/sca-j/download.php/30880/sca-javacaa-1.1-spec-cd02.doc>
<http://www.oasis-open.org/apps/org/workgroup/sca-j/download.php/31427/sca-javacaa-1.1-spec-cd02.pdf> (Authoritative)

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> (Authoritative)

Latest Approved Version:

Technical Committee:

[OASIS Service Component Architecture / J \(SCA-J\) TC](#)

Chair(s):

David Booz,	IBM
Mark Combellack,	Avaya

Editor(s):

David Booz,	IBM
Mark Combellack,	Avaya
Mike Edwards,	IBM
Anish Karmarkar,	Oracle

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/200912>

Abstract:

The SCA-J Common Annotations and APIs specification defines a Java syntax for programming concepts defined in the SCA Assembly Model Specification. It specifies a set of APIs and annotations that can be used by Java-based artifacts described by other SCA specifications such as [the POJO Component Implementation Specification \[JAVA_CI\]](#).

Specifically, this specification covers:

1. Implementation metadata for specifying component services, references, and properties
2. A client and component API
3. Metadata for asynchronous 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 other Java-based SCA specifications can choose 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 might 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, 2010. 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	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	10
2.2.2	Composite Scope	11
2.3	@AllowsPassByReference	11
2.3.1	Marking Services and References as “allows pass by reference”	12
2.3.2	Applying “allows pass by reference” to Service Proxies.....	12
2.3.3	Using “allows pass by reference” to Optimize Remotable Calls	13
3	Interface.....	14
3.1	Java Interface Element – <interface.java>	14
3.2	@Remotable	15
3.3	@Callback.....	15
3.4	@AsyncInvocation.....	15
3.5	SCA Java Annotations for Interface Classes.....	16
3.6	Compatibility of Java Interfaces	16
4	SCA Component Implementation Lifecycle	17
4.1	Overview of SCA Component Implementation Lifecycle	17
4.2	SCA Component Implementation Lifecycle State Diagram	17
4.2.1	Constructing State	18
4.2.2	Injecting State	18
4.2.3	Initializing State	19
4.2.4	Running State	19
4.2.5	Destroying State.....	19
4.2.6	Terminated State	20
5	Client API.....	21
5.1	Accessing Services from an SCA Component.....	21
5.1.1	Using the Component Context API	21
5.2	Accessing Services from non-SCA Component Implementations	21
5.2.1	SCAClientFactory Interface and Related Classes	21
6	Error Handling	23
7	Asynchronous Programming	24
7.1	@OneWay.....	24
7.2	Callbacks.....	24

7.2.1	Using Callbacks.....	24
7.2.2	Callback Instance Management.....	26
7.2.3	Callback Injection.....	26
7.2.4	Implementing Multiple Bidirectional Interfaces.....	26
7.2.5	Accessing Callbacks.....	27
7.3	Asynchronous handling of Long Running Service Operations.....	28
7.3.1	SCA Asynchronous Service Interface.....	28
8	Policy Annotations for Java.....	31
8.1	General Intent Annotations.....	31
8.2	Specific Intent Annotations.....	33
8.2.1	How to Create Specific Intent Annotations.....	33
8.3	Application of Intent Annotations.....	34
8.3.1	Intent Annotation Examples.....	34
8.3.2	Inheritance and Annotation.....	36
8.4	Relationship of Declarative and Annotated Intents.....	37
8.5	Policy Set Annotations.....	37
8.6	Security Policy Annotations.....	39
8.7	Transaction Policy Annotations.....	40
9	Java API.....	41
9.1	Component Context.....	41
9.2	Request Context.....	46
9.3	ServiceReference.....	49
9.4	ServiceRuntimeException.....	50
9.5	ServiceUnavailableException.....	51
9.6	InvalidServiceException.....	51
9.7	Constants.....	52
9.8	SCAClientFactory Class.....	52
9.9	SCAClientFactoryFinder Interface.....	56
9.10	SCAClientFactoryFinderImpl Class.....	56
9.11	NoSuchDomainException.....	57
9.12	NoSuchServiceException.....	57
10	Java Annotations.....	59
10.1	@AllowsPassByReference.....	59
10.2	@AsyncInvocation.....	60
10.3	@Authentication.....	61
10.4	@Authorization.....	62
10.5	@Callback.....	62
10.6	@ComponentName.....	64
10.7	@Confidentiality.....	64
10.8	@Constructor.....	65
10.9	@Context.....	66
10.10	@Destroy.....	66
10.11	@EagerInit.....	67
10.12	@Init.....	67
10.13	@Integrity.....	68

10.14	@Intent	68
10.15	@ManagedSharedTransaction	69
10.16	@ManagedTransaction.....	70
10.17	@MutualAuthentication.....	70
10.18	@NoManagedTransaction	71
10.19	@OneWay.....	72
10.20	@PolicySets	72
10.21	@Property	73
10.22	@Qualifier	74
10.23	@Reference	75
10.23.1	Reinjection	77
10.24	@Remotable	79
10.25	@Requires	81
10.26	@Scope	81
10.27	@Service.....	82
11	WSDL to Java and Java to WSDL.....	84
11.1	JAX-WS Client Asynchronous API for a Synchronous Service	87
12	Conformance	90
12.1	SCA Java XML Document	90
12.2	SCA Java Class.....	90
12.3	SCA Runtime.....	90
A.	XML Schema: sca-interface-java.xsd.....	91
B.	Java Classes and Interfaces.....	92
B.1	SCAClient Classes and Interfaces.....	92
B.1.1	SCAClientFactory Class.....	92
B.1.2	SCAClientFactoryFinder interface	94
B.1.3	SCAClientFactoryFinderImpl class	95
B.1.4	SCAClient Classes and Interfaces - what does a vendor need to do?	100
C.	Conformance Items.....	102
D.	Acknowledgements	116
E.	Non-Normative Text	118
F.	Revision History	119

1 Introduction

The SCA-J Common Annotations and APIs specification defines a Java syntax for programming concepts defined in the SCA Assembly Model Specification [ASSEMBLY]. It specifies a set of APIs and annotations that can be used by SCA Java-based 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 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

The goal of defining the annotations and APIs in this specification is to promote consistency and reduce duplication across the various SCA Java-based specifications. The annotations and APIs defined in this specification are designed to be used by other SCA Java-based specifications in either a partial or complete fashion.

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 Model Specification Version 1.1, http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd03.pdf |
| [JAVA_CI] | SCA POJO Component Implementation Specification Version 1.1 http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-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/wsd/ , |
| [POLICY] | SCA Policy Framework Version 1.1, http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd02.pdf |
| [JSR-250] | Common Annotations for the Java Platform specification (JSR-250), http://www.jcp.org/en/jsr/detail?id=250 |
| [JAX-WS] | JAX-WS 2.1 Specification (JSR-224), http://www.jcp.org/en/jsr/detail?id=224 |
| [JAVABEANS] | JavaBeans 1.01 Specification, http://java.sun.com/javase/technologies/desktop/javabeans/api/ |

44 **[JAAS]** Java Authentication and Authorization Service Reference Guide
45 [http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.](http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html)
46 [html](http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html)

47 **1.3 Non-Normative References**

48 **[EBNF-Syntax]** Extended BNF syntax format used for formal grammar of constructs
49 <http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation>

50 2 Implementation Metadata

51 This section describes SCA Java-based metadata, which applies to Java-based implementation
52 types.

53 2.1 Service Metadata

54 2.1.1 @Service

55 The **@Service annotation** is used on a Java class to specify the interfaces of the services provided
56 by the implementation. Service interfaces are defined in one of the following ways:

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

62 2.1.2 Java Semantics of a Remotable Service

63 A **remotable service** is defined using the @Remotable annotation on the Java interface or Java
64 class that defines the service, or on a service reference. Remotable services are intended to be
65 used for **coarse grained** services, and the parameters are passed **by-value**. **Remotable Services**
66 **MUST NOT make use of method overloading.** [JCA20001]

67 The following snippet shows an example of a Java interface for a remotable service:

```
68 package services.hello;  
69 @Remotable  
70 public interface HelloService {  
71     String hello(String message);  
72 }
```

73 2.1.3 Java Semantics of a Local Service

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

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

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

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

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

85 The data exchange semantic for calls to local services is **by-reference**. This means that
86 implementation code which uses a local interface needs to be written with the knowledge that
87 changes made to parameters (other than simple types) by either the client or the provider of the
88 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, or a
91 constructor parameter typed by the service interface and annotated with a **@Reference**
92 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 Model specification \[ASSEMBLY\]](#). The **@Property** annotation is used to define
96 an SCA 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 specification defines two scopes:

- 105 • STATELESS
- 106 • COMPOSITE

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

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

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

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

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

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

```
119  
120     @Init  
121     public void start() {  
122         ...  
123     }  
124  
125     @Destroy  
126     public void stop() {  
127         ...  
128     }  
129
```

130 The following sections specify the two standard scopes which a Java-based implementation type
131 can support.

132 2.2.1 Stateless Scope

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

135 The concurrency model for the stateless scope is single threaded. This means that the SCA
136 runtime MUST ensure that a stateless scoped implementation instance object is only ever
137 dispatched on one thread at any one time. [JCA20002] In addition, within the SCA lifecycle of a
138 stateless scoped implementation instance, the SCA runtime MUST only make a single invocation of
139 one business method. [JCA20003] Note that the SCA lifecycle might not correspond to the Java
140 object lifecycle due to runtime techniques such as pooling.

141 2.2.2 Composite Scope

142 The meaning of "composite scope" is defined in relation to the composite containing the
143 component.

144 It is important to distinguish between different uses of a composite, where these uses affect the
145 numbers of instances of components within the composite. There are 2 cases:

- 146 a) Where the composite containing the component using the Java implementation is the SCA
147 Domain (i.e. a deployment composite declares the component using the implementation)
- 148 b) Where the composite containing the component using the Java implementation is itself used
149 as the implementation of a higher level component (any level of nesting is possible, but the
150 component is NOT at the Domain level)

151 Where an implementation is used by a "domain level component", and the implementation is
152 marked "Composite" scope, the SCA runtime MUST ensure that all consumers of the component
153 appear to be interacting with a single runtime instance of the implementation. [JCA20004]

154 Where an implementation is marked "Composite" scope and it is used by a component that is
155 nested inside a composite that is used as the implementation of a higher level component, the
156 SCA runtime MUST ensure that all consumers of the component appear to be interacting with a
157 single runtime instance of the implementation. There can be multiple instances of the higher level
158 component, each running on different nodes in a distributed SCA runtime. [JCA20008]

159 The SCA runtime can exploit shared state technology in combination with other well known high
160 availability techniques to provide the appearance of a single runtime instance for consumers of
161 composite scoped components.

162 The lifetime of the containing composite is defined as the time it becomes active in the runtime to
163 the time it is deactivated, either normally or abnormally.

164 When the implementation class is marked for eager initialization, the SCA runtime MUST create a
165 composite scoped instance when its containing component is started. [JCA20005] If a method of
166 an implementation class is marked with the @Init annotation, the SCA runtime MUST call that
167 method when the implementation instance is created. [JCA20006]

168 The concurrency model for the composite scope is multi-threaded. This means that the SCA
169 runtime MAY run multiple threads in a single composite scoped implementation instance object
170 and the SCA runtime MUST NOT perform any synchronization. [JCA20007]

171 2.3 @AllowsPassByReference

172 Calls to remotable services (see [section "Java Semantics of a Remotable Service"](#)) have by-value
173 semantics. This means that input parameters passed to the service can be modified by the
174 service without these modifications being visible to the client. Similarly, the return value or
175 exception from the service can be modified by the client without these modifications being visible
176 to the service implementation. For remote calls (either cross-machine or cross-process), these
177 semantics are a consequence of marshalling input parameters, return values and exceptions "on
178 the wire" and unmarshalling them "off the wire" which results in physical copies being made. For
179 local method calls within the same JVM, Java language calling semantics are by-reference and
180 therefore do not provide the correct by-value semantics for SCA remotable interfaces. To
181 compensate for this, the SCA runtime can intervene in these calls to provide by-value semantics
182 by making copies of any mutable objects passed.

183 The cost of such copying can be very high relative to the cost of making a local call, especially if
184 the data being passed is large. Also, in many cases this copying is not needed if the

185 implementation observes certain conventions for how input parameters, return values and
186 exceptions are used. The `@AllowsPassByReference` annotation allows service method
187 implementations and client references to be marked as "allows pass by reference" to indicate that
188 they use input parameters, return values and exceptions in a manner that allows the SCA runtime
189 to avoid the cost of copying mutable objects when a remotable service is called locally within the
190 same JVM.

191 **2.3.1 Marking Services and References as "allows pass by reference"**

192 Marking a service method implementation as "allows pass by reference" asserts that the method
193 implementation observes the following restrictions:

- 194 • Method execution will not modify any input parameter before the method returns.
- 195 • The service implementation will not retain a reference to any mutable input parameter,
196 mutable return value or mutable exception after the method returns.
- 197 • The method will observe "allows pass by value" client semantics (see below) for any
198 callbacks that it makes.

199 See [section "@AllowsPassByReference"](#) for details of how the `@AllowsPassByReference` annotation
200 is used to mark a service method implementation as "allows pass by reference".

201 Marking a client reference as "allows pass by reference" asserts that method calls through the
202 reference observe the following restrictions:

- 203 • The client implementation will not modify any of the method's input parameters before
204 the method returns. Such modifications might occur in callbacks or separate client
205 threads.
- 206 • If the method is one-way, the client implementation will not modify any of the method's
207 input parameters at any time after calling the method. This is because one-way method
208 calls return immediately without waiting for the service method to complete.

209 See [section "Applying "allows pass by reference" to Service Proxies"](#) for details of how the
210 `@AllowsPassByReference` annotation is used to mark a client reference as "allows pass by
211 reference".

212 **2.3.2 Applying "allows pass by reference" to Service Proxies**

213 Service method calls are made by clients using service proxies, which can be obtained by injection
214 into client references or by making API calls. A service proxy is marked as "allows pass by
215 reference" if and only if any of the following applies:

- 216 • It is injected into a reference or callback reference that is marked "allows pass by
217 reference".
- 218 • It is obtained by calling `ComponentContext.getService()` or
219 `ComponentContext.getServices()` with the name of a reference that is marked "allows
220 pass by reference".
- 221 • It is obtained by calling `RequestContext.getCallback()` from a service implementation that
222 is marked "allows pass by reference".
- 223 • It is obtained by calling `ServiceReference.getService()` on a service reference that is
224 marked "allows pass by reference" (see definition below).

225 A service reference for a remotable service call is marked "allows pass by reference" if and only if
226 any of the following applies:

- 227 • It is injected into a reference or callback reference that is marked "allows pass by
228 reference".
- 229 • It is obtained by calling `ComponentContext.getServiceReference()` or
230 `ComponentContext.getServiceReferences()` with the name of a reference that is marked
231 "allows pass by reference".

- 232
- 233
- It is obtained by calling `RequestContext.getCallbackReference()` from a service implementation that is marked "allows pass by reference".
- 234
- It is obtained by calling `ComponentContext.cast()` on a proxy that is marked "allows pass by reference".
- 235

236

2.3.3 Using "allows pass by reference" to Optimize Remotable Calls

237

238

239

240

The SCA runtime MAY use by-reference semantics when passing input parameters, return values or exceptions on calls to remotable services within the same JVM if both the service method implementation and the service proxy used by the client are marked "allows pass by reference". [JCA20009]

241

242

243

244

The SCA runtime MUST use by-value semantics when passing input parameters, return values and exceptions on calls to remotable services within the same JVM if the service method implementation is not marked "allows pass by reference" or the service proxy used by the client is not marked "allows pass by reference". [JCA20010]

245 3 Interface

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

247 3.1 Java Interface Element – <interface.java>

248 The Java interface element is used in SCA Documents in places where an interface is declared in
249 terms of a Java interface class. The Java interface element identifies the Java interface class and
250 can also identify a callback interface, where the first Java interface represents the forward
251 (service) call interface and the second interface represents the interface used to call back from the
252 service to the client.

253 It is possible that the Java interface class referenced by the <interface.java/> element contains
254 one or more annotations defined by the JAX-WS specification [JAX-WS]. These annotations can
255 affect the interpretation of the <interface.java/> element. In the most extreme case, the
256 annotations cause the replacement of the <interface.java/> element with an <interface.wsdl/>
257 element. The relevant JAX-WS annotations and their effects on the <interface.java/> element are
258 described in the section "[JAX-WS Annotations and SCA Interfaces](#)".

259 The `interface.java` element MUST conform to the schema defined in the `sca-interface-java.xsd`
260 schema. [JCA30004]

261 The following is the pseudo-schema for the `interface.java` element

262

```
263 <interface.java interface="NCName" callbackInterface="NCName"?  
264     requires="list of xs:QName"?  
265     policySets="list of xs:QName"?  
266     remotable="boolean"?/>
```

267

268 The `interface.java` element has the following attributes:

- 269 • **interface : NCName (1..1)** – the Java interface class to use for the service interface. The
270 value of the `@interface` attribute MUST be the fully qualified name of the Java interface
271 class [JCA30001]
272 If the identified class is annotated with either the JAX-WS `@WebService` or
273 `@WebServiceProvider` annotations and the annotation has a non-empty **wsdlLocation**
274 property, then the SCA Runtime MUST act as if an `<interface.wsdl/>` element is present
275 instead of the `<interface.java/>` element, with an `@interface` attribute identifying the portType
276 mapped from the Java interface class and containing `@requires` and `@policySets` attribute
277 values equal to the `@requires` and `@policySets` attribute values of the `<interface.java/>`
278 element. [JCA30010]
- 279 • **callbackInterface : NCName (0..1)** – the Java interface class to use for the callback
280 interface. The value of the `@callbackInterface` attribute MUST be the fully qualified name
281 of a Java interface used for callbacks [JCA30002]
- 282 • **requires : QName (0..1)** – a list of policy intents. See the [Policy Framework specification](#)
283 [POLICY] for a description of this attribute
- 284 • **policySets : QName (0..1)** – a list of policy sets. See the [Policy Framework specification](#)
285 [POLICY] for a description of this attribute.
- 286 • **remotable : boolean (0..1)** – indicates whether or not the interface is remotable. A value of
287 “true” means the interface is remotable and a value of “false” means it is not. This attribute
288 does not have a default value. If it is not specified then the remotability is determined by the
289 presence or absence of the `@Remotable` annotation on the interface class. The `@remotable`
290 attribute applies to both the interface and any optional `callbackInterface`. The `@remotable`
291 attribute is intended as an alternative to using the `@Remotable` annotation on the interface

292 class. The value of the @remotable attribute on the <interface.java/> element does not
293 override the presence of a @Remotable annotation on the interface class and so if the
294 interface class contains a @Remotable annotation and the @remotable attribute has a
295 value of "false", then the SCA Runtime MUST raise an error and MUST NOT run the
296 component concerned. [JCA30005]

297

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

299

```
300 <interface.java interface="services.stockquote.StockQuoteService"  
301     callbackInterface="services.stockquote.StockQuoteServiceCallback"/>  
302
```

303 Here, the Java interface is defined in the Java class file
304 `./services/stockquote/StockQuoteService.class`, where the root directory is defined by the
305 contribution in which the interface exists. Similarly, the callback interface is defined in the Java
306 class file `./services/stockquote/StockQuoteServiceCallback.class`.

307 Note that the Java interface class identified by the @interface attribute can contain a Java
308 @Callback annotation which identifies a callback interface. If this is the case, then it is not
309 necessary to provide the @callbackInterface attribute. However, if the Java interface class
310 identified by the @interface attribute does contain a Java @Callback annotation, then the Java
311 interface class identified by the @callbackInterface attribute MUST be the same interface class.
312 [JCA30003]

313 For the Java interface type system, parameters and return types of the service methods are
314 described using Java classes or simple Java types. It is recommended that the Java Classes used
315 conform to the requirements of either JAXB [JAX-B] or of Service Data Objects [SDO] because of
316 their integration with XML technologies.

317 3.2 @Remotable

318 The **@Remotable** annotation on a Java interface, a service implementation class, or a service
319 reference denotes an interface or class that is designed to be used for remote communication.
320 Remotable interfaces are intended to be used for **coarse grained** services. Operations'
321 parameters, return values and exceptions are passed **by-value**. Remotable Services are not
322 allowed to make use of method **overloading**.

323 3.3 @Callback

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

328 3.4 @AsyncInvocation

329 An interface can be annotated with @AsyncInvocation or with the equivalent
330 @Requires("sca:asyncInvocation") annotation to indicate that request/response operations of that
331 interface are **long running** and that response messages are likely to be sent an arbitrary length
332 of time after the initial request message is sent to the target service. This is described in the [SCA
333 Assembly Specification \[ASSEMBLY\]](#).

334 For a service client, it is strongly recommended that the client uses the asynchronous form of the
335 client interface when using a reference to a service with an interface annotated with
336 @AsyncInvocation, using either polling or callbacks to receive the response message. See the
337 sections "[Asynchronous Programming](#)" and the section "[JAX-WS Client Asynchronous API for a
338 Synchronous Service](#)" for more details about the asynchronous client API.

339 For a service implementation, SCA provides an **asynchronous service** mapping of the WSDL
340 request/response interface which enables the service implementation to send the response
341 message at an arbitrary time after the original service operation is invoked. This is described in
342 the section "[Asynchronous handling of Long Running Service Operations](#)".

343 **3.5 SCA Java Annotations for Interface Classes**

344 A Java interface referenced by the @interface attribute of an <interface.java/> element MUST NOT
345 contain any of the following SCA Java annotations:

346 @AllowsPassByReference, @ComponentName, @Constructor, @Context, @Destroy, @EagerInit,
347 @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service. [JCA30006]

348 A Java interface referenced by the @callbackInterface attribute of an <interface.java/> element MUST
349 NOT contain any of the following SCA Java annotations:

350 @AllowsPassByReference, @Callback, @ComponentName, @Constructor, @Context, @Destroy,
351 @EagerInit, @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service. [JCA30007]

352 **3.6 Compatibility of Java Interfaces**

353 The SCA Assembly Model specification [ASSEMBLY] defines a number of criteria that need to be
354 satisfied in order for two interfaces to be compatible or have a compatible superset or subset
355 relationship. If these interfaces are both Java interfaces, compatibility also means that every
356 method that is present in both interfaces is defined consistently in both interfaces with respect to
357 the @OneWay annotation, that is, the annotation is either present in both interfaces or absent in
358 both interfaces. [JCA30009]

359 4 SCA Component Implementation Lifecycle

360 This section describes the lifecycle of an SCA component implementation.

361 4.1 Overview of SCA Component Implementation Lifecycle

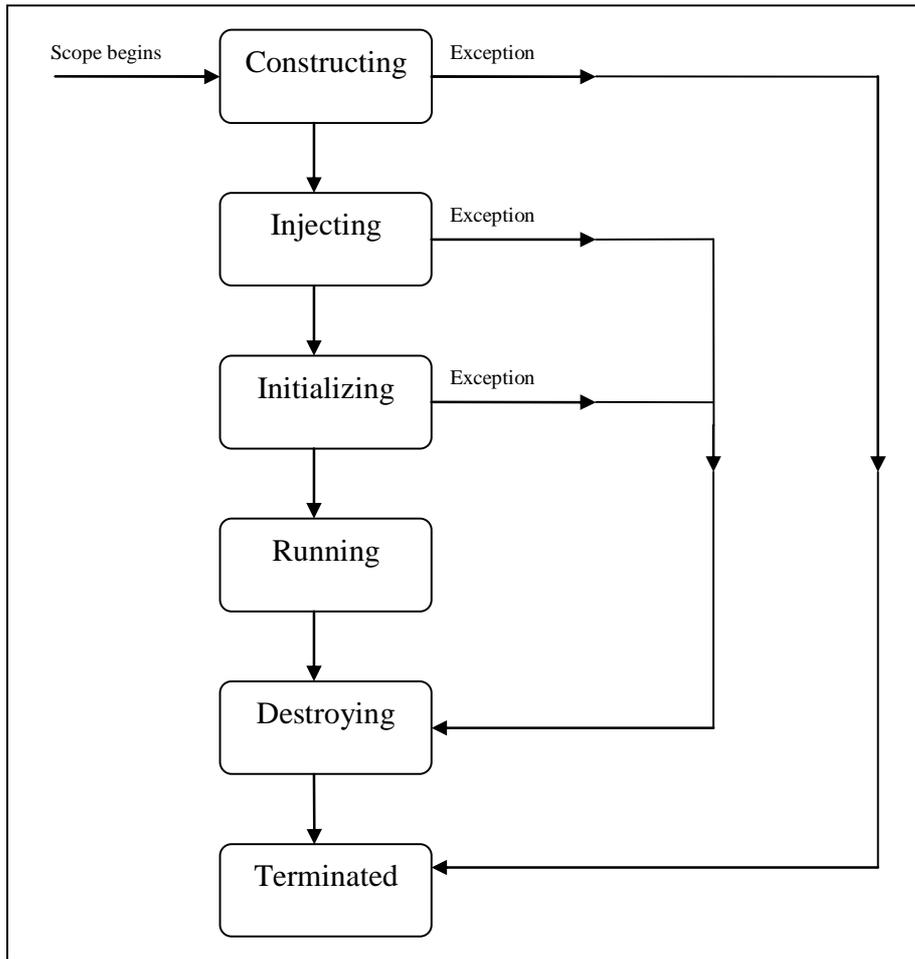
362 At a high level, there are 3 main phases through which an SCA component implementation will
363 transition when it is used by an SCA Runtime:

- 364 1. **The Initialization phase.** This involves constructing an instance of the component
365 implementation class and injecting any properties and references. Once injection is
366 complete, the method annotated with @Init is called, if present, which provides the
367 component implementation an opportunity to perform any internal initialization it requires.
- 368 2. **The Running phase.** This is where the component implementation has been initialized
369 and the SCA Runtime can dispatch service requests to it over its Service interfaces.
- 370 3. **The Destroying phase.** This is where the component implementation's scope has ended
371 and the SCA Runtime destroys the component implementation instance. The SCA Runtime
372 calls the method annotated with @Destroy, if present, which provides the component
373 implementation an opportunity to perform any internal clean up that is required.

374 4.2 SCA Component Implementation Lifecycle State Diagram

375 The state diagram in Figure 4.1 shows the lifecycle of an SCA component implementation. The
376 sections that follow it describe each of the states that it contains.

377 It should be noted that some component implementation specifications might not implement all
378 states of the lifecycle. In this case, that state of the lifecycle is skipped over.



379
380
381

Figure 4.1 SCA - Component implementation lifecycle

4.2.1 Constructing State

382
383 The SCA Runtime MUST call a constructor of the component implementation at the start of the
384 Constructing state. [JCA40001] The SCA Runtime MUST perform any constructor reference or
385 property injection when it calls the constructor of a component implementation. [JCA40002]

386 The result of invoking operations on any injected references when the component implementation
387 is in the Constructing state is undefined.

388 When the constructor completes successfully, the SCA Runtime MUST transition the component
389 implementation to the Injecting state. [JCA40003] If an exception is thrown whilst in the
390 Constructing state, the SCA Runtime MUST transition the component implementation to the
391 Terminated state. [JCA40004]

4.2.2 Injecting State

392
393 When a component implementation instance is in the Injecting state, the SCA Runtime MUST first
394 inject all field and setter properties that are present into the component implementation.
395 [JCA40005] The order in which the properties are injected is unspecified.

396 When a component implementation instance is in the Injecting state, the SCA Runtime MUST
397 inject all field and setter references that are present into the component implementation, after all

398 the properties have been injected. [JCA40006] The order in which the references are injected is
399 unspecified.

400 The SCA Runtime MUST ensure that the correct synchronization model is used so that all injected
401 properties and references are made visible to the component implementation without requiring the
402 component implementation developer to do any specific synchronization. [JCA40007]

403 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the
404 component implementation is in the Injecting state. [JCA40008]

405 The result of invoking operations on any injected references when the component implementation
406 is in the Injecting state is undefined.

407 When the injection of properties and references completes successfully, the SCA Runtime MUST
408 transition the component implementation to the Initializing state. [JCA40009] If an exception is
409 thrown whilst injecting properties or references, the SCA Runtime MUST transition the component
410 implementation to the Destroying state. [JCA40010] If a property or reference is unable to be
411 injected, the SCA Runtime MUST transition the component implementation to the Destroying
412 state. [JCA40024]

413 4.2.3 Initializing State

414 When the component implementation enters the Initializing State, the SCA Runtime MUST call the
415 method annotated with @Init on the component implementation, if present. [JCA40011]

416 The component implementation can invoke operations on any injected references when it is in the
417 Initializing state. However, depending on the order in which the component implementations are
418 initialized, the target of the injected reference might not be available since it has not yet been
419 initialized. If a component implementation invokes an operation on an injected reference that
420 refers to a target that has not yet been initialized, the SCA Runtime MUST throw a
421 ServiceUnavailableException. [JCA40012]

422 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the
423 component implementation instance is in the Initializing state. [JCA40013]

424 Once the method annotated with @Init completes successfully, the SCA Runtime MUST transition
425 the component implementation to the Running state. [JCA40014] If an exception is thrown whilst
426 initializing, the SCA Runtime MUST transition the component implementation to the Destroying
427 state. [JCA40015]

428 4.2.4 Running State

429 The SCA Runtime MUST invoke Service methods on a component implementation instance when
430 the component implementation is in the Running state and a client invokes operations on a service
431 offered by the component. [JCA40016]

432 The component implementation can invoke operations on any injected references when the
433 component implementation instance is in the Running state.

434 When the component implementation scope ends, the SCA Runtime MUST transition the
435 component implementation to the Destroying state. [JCA40017]

436 4.2.5 Destroying State

437 When a component implementation enters the Destroying state, the SCA Runtime MUST call the
438 method annotated with @Destroy on the component implementation, if present. [JCA40018]

439 The component implementation can invoke operations on any injected references when it is in the
440 Destroying state. However, depending on the order in which the component implementations are
441 destroyed, the target of the injected reference might no longer be available since it has been
442 destroyed. If a component implementation invokes an operation on an injected reference that
443 refers to a target that has been destroyed, the SCA Runtime MUST throw an
444 InvalidServiceException. [JCA40019]

445 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the
446 component implementation instance is in the Destroying state. [JCA40020]

447 Once the method annotated with @Destroy completes successfully, the SCA Runtime MUST
448 transition the component implementation to the Terminated state. [JCA40021] If an exception is
449 thrown whilst destroying, the SCA Runtime MUST transition the component implementation to the
450 Terminated state. [JCA40022]

451 **4.2.6 Terminated State**

452 The lifecycle of the SCA Component has ended.

453 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the
454 component implementation instance is in the Terminated state. [JCA40023]

455 5 Client API

456 This section describes how SCA services can be programmatically accessed from components and
457 also from non-managed code, that is, code not running as an SCA component.

458 5.1 Accessing Services from an SCA Component

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

463 5.1.1 Using the Component Context API

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

467 5.2 Accessing Services from non-SCA Component Implementations

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

470 5.2.1 SCAClientFactory Interface and Related Classes

471 Client code can use the **SCAClientFactory** class to obtain proxy reference objects for a service
472 which is in an SCA Domain. The URI of the domain, the relative URI of the service and the
473 business interface of the service must all be known in order to use the SCAClientFactory class.

474
475 Objects which implement the SCAClientFactory are obtained using the newInstance() methods of
476 the SCAClientFactory class.

477 The following is a sample of the code that a client would use:

```
478 package org.oasisopen.sca.client.example;  
479  
480 import java.net.URI;  
481  
482 import org.oasisopen.sca.client.SCAClientFactory;  
483 import org.oasisopen.sca.client.example.HelloService;  
484  
485 /**  
486  * Example of use of Client API for a client application to obtain  
487  * an SCA reference proxy for a service in an SCA Domain.  
488  */  
489 public class Client1 {  
490  
491     public void someMethod() {  
492  
493         try {  
494  
495             String serviceURI = "SomeHelloServiceURI";  
496             URI domainURI = new URI("SomeDomainURI");  
497  
498             SCAClientFactory scaClient =  
499                 SCAClientFactory.newInstance( domainURI );  
500             HelloService helloService =
```

```
501         scaClient.getService(HelloService.class,  
502                               serviceURI);  
503         String reply = helloService.sayHello("Mark");  
504  
505         } catch (Exception e) {  
506             System.out.println("Received exception");  
507         }  
508     }  
509 }
```

511 For details about the SCAClientFactory interface and its related classes see the section
512 ["SCAClientFactory Class"](#).

513

514 **6 Error Handling**

515 Clients calling service methods can experience business exceptions and SCA runtime exceptions.

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

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

521 7 Asynchronous Programming

522 Asynchronous programming of a service is where a client invokes a service and carries on
523 executing without waiting for the service to execute. Typically, the invoked service executes at
524 some later time. Output from the invoked service, if any, is fed back to the client through a
525 separate mechanism, since no output is available at the point where the service is invoked. This is
526 in contrast to the call-and-return style of synchronous programming, where the invoked service
527 executes and returns any output to the client before the client continues. The SCA asynchronous
528 programming model consists of:

- 529 • support for non-blocking method calls
- 530 • callbacks

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

532 7.1 @OneWay

533 **Non-blocking calls** represent the simplest form of asynchronous programming, where the client
534 of the service invokes the service and continues processing immediately, without waiting for the
535 service to execute.

536 A method with a void return type and which has no declared exceptions can be marked with a
537 **@OneWay** annotation. This means that the method is non-blocking and communication with the
538 service provider can use a binding that buffers the request and sends it at some later time.

539 For a Java client to make a non-blocking call to methods that either return values or throw
540 exceptions, a Java client can use the JAX-WS asynchronous client API model that is described in
541 [the section "JAX-WS Client Asynchronous API for a Synchronous Service"](#). It is considered to be a
542 best practice that service designers define one-way methods as often as possible, in order to give
543 the greatest degree of binding flexibility to deployers.

544 7.2 Callbacks

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

- 549 • an interface for the provided service
- 550 • a callback interface that is provided by the client

551 Callbacks can be used for both remotable and local services. Either both interfaces of a
552 bidirectional service are remotable, or both are local. It is illegal to mix the two, as defined in [the](#)
553 [SCA Assembly Model specification \[ASSEMBLY\]](#).

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

558 7.2.1 Using Callbacks

559 Bidirectional interfaces and callbacks are used when a simple request/response pattern isn't
560 sufficient to capture the business semantics of a service interaction. Callbacks are well suited for
561 cases when a service request can result in multiple responses or new requests from the service
562 back to the client, or where the service might respond to the client some time after the original
563 request has completed.

564 The following example shows a scenario in which bidirectional interfaces and callbacks could be
565 used. A client requests a quotation from a supplier. To process the enquiry and return the
566 quotation, some suppliers might need additional information from the client. The client does not
567 know which additional items of information will be needed by different suppliers. This interaction
568 can be modeled as a bidirectional interface with callback requests to obtain the additional
569 information.

```
570 package somepackage;
571 import org.oasisopen.sca.annotation.Callback;
572 import org.oasisopen.sca.annotation.Remotable;
573
574 @Remotable
575 @Callback(QuotationCallback.class)
576 public interface Quotation {h
577     double requestQuotation(String productCode, int quantity);
578 }
579
580 @Remotable
581 public interface QuotationCallback {
582     String getState();
583     String getZipCode();
584     String getCreditRating();
585 }
586
```

587 In this example, the `requestQuotation` operation requests a quotation to supply a given quantity
588 of a specified product. The `QuotationCallback` interface provides a number of operations that the
589 supplier can use to obtain additional information about the client making the request. For
590 example, some suppliers might quote different prices based on the state or the ZIP code to which
591 the order will be shipped, and some suppliers might quote a lower price if the ordering company
592 has a good credit rating. Other suppliers might quote a standard price without requesting any
593 additional information from the client.

594 The following code snippet illustrates a possible implementation of the example service, using the
595 `@Callback` annotation to request that a callback proxy be injected.

```
596 @Callback
597 protected QuotationCallback callback;
598
599 public double requestQuotation(String productCode, int quantity) {
600     double price = getPrice(productCode, quantity);
601     double discount = 0;
602     if (quantity > 1000 && callback.getState().equals("FL")) {
603         discount = 0.05;
604     }
605     if (quantity > 10000 && callback.getCreditRating().charAt(0) == 'A') {
606         discount += 0.05;
607     }
608     return price * (1-discount);
609 }
610
611
```

612 The code snippet below is taken from the client of this example service. The client's service
613 implementation class implements the methods of the `QuotationCallback` interface as well as those
614 of its own service interface `ClientService`.

```
615 public class ClientImpl implements ClientService, QuotationCallback {
616     private QuotationService myService;
617
618
619
```

```

620     @Reference
621     public void setMyService(QuotationService service) {
622         myService = service;
623     }
624
625     public void aClientMethod() {
626         ...
627         double quote = myService.requestQuotation("AB123", 2000);
628         ...
629     }
630
631     public String getState() {
632         return "TX";
633     }
634     public String getZipCode() {
635         return "78746";
636     }
637     public String getCreditRating() {
638         return "AA";
639     }
640 }

```

642 In this example the callback is **stateless**, i.e., the callback requests do not need any information
643 relating to the original service request. For a callback that needs information relating to the
644 original service request (a **stateful** callback), this information can be passed to the client by the
645 service provider as parameters on the callback request.

646 7.2.2 Callback Instance Management

647 Instance management for callback requests received by the client of the bidirectional service is
648 handled in the same way as instance management for regular service requests. If the client
649 implementation has STATELESS scope, the callback is dispatched using a newly initialized
650 instance. If the client implementation has COMPOSITE scope, the callback is dispatched using the
651 same shared instance that is used to dispatch regular service requests.

652 As described in [the section "Using Callbacks"](#), a stateful callback can obtain information relating to
653 the original service request from parameters on the callback request. Alternatively, a composite-
654 scoped client could store information relating to the original request as instance data and retrieve
655 it when the callback request is received. These approaches could be combined by using a key
656 passed on the callback request (e.g., an order ID) to retrieve information that was stored in a
657 composite-scoped instance by the client code that made the original request.

658 7.2.3 Callback Injection

659 **When a bidirectional service is invoked, the SCA runtime MUST inject a callback reference for the**
660 **invoking service into all fields and setter methods of the service implementation class that are**
661 **marked with a @Callback annotation and typed by the callback interface of the bidirectional**
662 **service, and the SCA runtime MUST inject null into all other fields and setter methods of the**
663 **service implementation class that are marked with a @Callback annotation. [JCA60001] When a**
664 **non-bidirectional service is invoked, the SCA runtime MUST inject null into all fields and setter**
665 **methods of the service implementation class that are marked with a @Callback annotation.**
666 [JCA60002]

667 7.2.4 Implementing Multiple Bidirectional Interfaces

668 Since it is possible for a single implementation class to implement multiple services, it is also
669 possible for callbacks to be defined for each of the services that it implements. The service
670 implementation can include an injected field for each of its callbacks. The runtime injects the
671 callback onto the appropriate field based on the type of the callback. The following shows the

672 declaration of two fields, each of which corresponds to a particular service offered by the
673 implementation.

```
674  
675 @Callback  
676 protected MyService1Callback callback1;  
677  
678 @Callback  
679 protected MyService2Callback callback2;
```

680

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

683 7.2.5 Accessing Callbacks

684 In addition to injecting a reference to a callback service, it is also possible to obtain a reference to
685 a Callback instance by annotating a field or method of type **ServiceReference** with the
686 **@Callback** annotation.

687

688 A reference implementing the callback service interface can be obtained using
689 `ServiceReference.getService()`.

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

```
691  
692 @Callback  
693 protected ServiceReference<MyCallback> callback;  
694  
695 public void someMethod() {  
696  
697     MyCallback myCallback = callback.getService();    ...  
698  
699     myCallback.receiveResult(theResult);  
700 }  
701
```

702 Because `ServiceReference` objects are serializable, they can be stored persistently and retrieved at
703 a later time to make a callback invocation after the associated service request has completed.
704 `ServiceReference` objects can also be passed as parameters on service invocations, enabling the
705 responsibility for making the callback to be delegated to another service.

706 Alternatively, a callback can be retrieved programmatically using the **RequestContext** API. The
707 snippet below shows how to retrieve a callback in a method programmatically:

```
708 @Context  
709 ComponentContext context;  
710  
711 public void someMethod() {  
712  
713     MyCallback myCallback =  
714         context.getRequestContext().getCallback();  
715  
716     ...  
717  
718     myCallback.receiveResult(theResult);  
719 }  
720
```

721

722 This is necessary if the service implementation has **COMPOSITE** scope, because callback injection
is not performed for composite-scoped implementations.

7.3 Asynchronous handling of Long Running Service Operations

Long-running request-response operations are described in the SCA Assembly Specification [ASSEMBLY]. These operations are characterized by following the WSDL request-response message exchange pattern, but where the timing of the sending of the response message is arbitrarily later than the receipt of the request message, with an impact on the client component, on the service component and also on the transport binding used to communicate between them.

In SCA, such operations are marked with an intent "asyncInvocation" and is expected that the client component, the service component and the binding are all affected by the presence of this intent. This specification does not describe the effects of the intent on the binding, other than to note that in general, there is an implication that the sending of the response message is typically separate from the sending of the request message, typically requiring a separate response endpoint on the client to which the response can be sent.

For components that are clients of a long-running request-response operation, it is strongly recommended that the client makes use of the JAX-WS Client Asynchronous API, either using the polling interface or the callback mechanism described in the section "[JAX-WS Client Asynchronous API for a Synchronous Service](#)". The principle is that the client should not synchronously wait for a response from the long running operation since this could take a long time and it is preferable not to tie up resources while waiting.

For the service implementation component, the JAX-WS client asynchronous API is not suitable, so the SCA Java Common Annotations and APIs specification defines the SCA Asynchronous Service interface, which, like the JAX-WS client asynchronous API, is an alternative mapping of a WSDL request-response operation into a Java interface.

7.3.1 SCA Asynchronous Service Interface

The SCA Asynchronous Service interface follows some of the patterns defined by the JAX-WS client asynchronous API, but it is a simpler interface aligned with the needs of a service implementation class.

As an example, for a WSDL portType with a single operation "getPrice" with a String request parameter and a float response, the synchronous Java interface mapping appears in snippet 7-1:

```
// synchronous mapping
public interface StockQuote {
    float getPrice(String ticker);
}
```

Snippet 7-1: Example synchronous Java interface mapping

The JAX-WS client asynchronous API for the same portType adds two asynchronous forms for each synchronous method, as shown in snippet 7-2:

```
// asynchronous mapping
public interface StockQuote {
    float getPrice(String ticker);
    Response<Float> getPriceAsync(String ticker);
    Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
}
```

Snippet 7-2: Example JAX-WS client asynchronous Java interface mapping

The SCA Asynchronous Service interface has a single method similar to the final one in the asynchronous client interface, as shown in snippet 7-2:

```
// asynchronous mapping
```

```

772 @Requires("sca:asyncInvocation")
773 public interface StockQuote {
774     void getPriceAsync(String ticker, ResponseDispatch<Float>);
775 }

```

776 *Snippet 7-3: Example SCA asynchronous service Java interface mapping*

777 The main characteristics of the SCA asynchronous mapping are:

- 778 • there is a single method, with a name with the string "Async" appended to the
- 779 operation name
- 780 • it has a void return type
- 781 • it has two input parameters, the first is the request message of the operation
- 782 and the second is a ResponseDispatch object typed by the response message of
- 783 the operation (following the rules expressed in the JAX-WS specification for the
- 784 typing of the AsyncHandler object in the client asynchronous API)
- 785 • it is annotated with the asyncInvocation intent
- 786 • if the synchronous method has any business faults/exceptions, it is annotated
- 787 with @AsyncFault, containing a list of the exception classes

788 Unlike the JAX-WS asynchronous client interface, there is only a single operation for the

789 service implementation to provide (it would be inconvenient for the service

790 implementation to be required to implement multiple methods for each operation in the

791 WSDL interface).

792 The ResponseDispatch parameter is the mechanism by which the service

793 implementation sends back the response message resulting from the invocation of the

794 service method. The ResponseDispatch is serializable and it can be invoked once at any

795 time after the invocation of the service method, either before or after the service

796 method returns. This enables the service implementation to store the

797 ResponseDispatch in serialized form and release resources while waiting for the

798 completion of whatever activities result from the processing of the initial invocation.

799 The ResponseDispatch object is allocated by the SCA runtime/binding implementation

800 and it is expected to contain whatever metadata is required to deliver the response

801 message back to the client that invoked the service operation.

802 **The SCA asynchronous service Java interface mapping of a WSDL request-response operation MUST**

803 **appear as follows:**

804 **The interface is annotated with the "asyncInvocation" intent.**

805 **For each service operation in the WSDL, the Java interface contains an operation with**

806 **- a name which is the JAX-WS mapping of the WSDL operation name, with the suffix "Async" added**

807 **- a void return type**

808 **- a set of input parameter(s) which match the JAX-WS mapping of the input parameter(s) of the**

809 **WSDL operation plus an additional last parameter which is a ResponseDispatch object typed by**

810 **the JAX-WS Response Bean mapping of the output parameter(s) of the WSDL operation, where**

811 **ResponseDispatch is the type defined in the SCA Java Common Annotations and APIs**

812 **specification. [JCA60003]**

813 **An SCA Runtime MUST support the use of the SCA asynchronous service interface for the interface**

814 **of an SCA service. [JCA60004].**

815 The ResponseDispatch object passed in as a parameter to a method of a service

816 implementation using the SCA asynchronous service Java interface can be invoked once

817 only through either its sendResponse method or through its sendFault method to return

818 the response resulting from the service method invocation. **If the SCA asynchronous**

819 service interface ResponseDispatch handleResponse method is invoked more than once through
820 either its sendResponse or its sendFault method, the SCA runtime MUST throw an
821 IllegalStateException. [JCA60005]

822

823 For the purposes of matching interfaces (when wiring between a reference and a service, or when using
824 an implementation class by a component), an interface which has one or more methods which follow the
825 SCA asynchronous service pattern MUST be treated as if those methods are mapped as the equivalent
826 synchronous methods, as follows:

827 Asynchronous service methods are characterized by:

- 828 a) void return type
- 829 b) a method name with the suffix "Async"
- 830 c) a last input parameter with a type of ResponseDispatch<X>
- 831 d) annotation with the asyncInvocation intent
- 832 e) possible annotation with the @AsyncFault annotation

833 The mapping of each such method is as if the method had the return type "X", the method name
834 without the suffix "Async" and all the input parameters except the last parameter of the type
835 ResponseDispatch<X>, plus the list of exceptions contained in the @AsyncFault annotation.
836 [JCA60006]

837

838 8 Policy Annotations for Java

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

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

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

856 This specification has a series of annotations which provide the capability for the developer to
857 attach policy information to Java implementation code. The annotations concerned first provide
858 general facilities for attaching SCA Intents and Policy Sets to Java code. Secondly, there are
859 further specific annotations that deal with particular policy intents for certain policy domains such
860 as Security and Transactions.

861 This specification supports using [the Common Annotations for the Java Platform specification \(JSR-
862 250\) \[JSR-250\]](#). An implication of adopting the common annotation for Java platform specification
863 is that the SCA Java specification supports consistent annotation and Java class inheritance
864 relationships. SCA policy annotation semantics follow the General Guidelines for Inheritance of
865 Annotations in [the Common Annotations for the Java Platform specification \[JSR-250\]](#), except that
866 member-level annotations in a class or interface do not have any effect on how class-level
867 annotations are applied to other members of the class or interface.

868

869 8.1 General Intent Annotations

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

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

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

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

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

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

```
882 public static final String SCA_PREFIX =  
883     "{http://docs.oasis-open.org/ns/opencsa/sca/200912}";  
884 public static final String CONFIDENTIALITY =  
885     SCA_PREFIX + "confidentiality";
```

```
886     public static final String CONFIDENTIALITY_MESSAGE =
887         CONFIDENTIALITY + ".message";
888
```

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

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

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

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

897

898 This attaches the intents "confidentiality.message" and "integrity.message".

899 The following is an example of a reference requiring support for confidentiality:

```
900     package com.foo;
901
902     import static org.oasisopen.sca.annotation.Confidentiality.*;
903     import static org.oasisopen.sca.annotation.Reference;
904     import static org.oasisopen.sca.annotation.Requires;
905
906     public class Foo {
907         @Requires(CONFIDENTIALITY)
908         @Reference
909         public void setBar(Bar bar) {
910             ...
911         }
912     }
913
```

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

```
917     package com.foo;
918
919     import static org.oasisopen.sca.Constants.*;
920     import static org.oasisopen.sca.annotation.Reference;
921     import static org.oasisopen.sca.annotation.Requires;
922
923     public class Foo {
924         @Requires(SCA_PREFIX+"confidentiality")
925         @Reference
926         public void setBar(Bar bar) {
927             ...
928         }
929     }
930
```

931 The formal syntax [EBNF-Syntax] for the @Requires annotation follows:

```
932     '@Requires("'" QualifiedIntent "'" (','" QualifiedIntent "'')* ')
```

933 where

```
934     QualifiedIntent ::= QName('.' Qualifier)*
935     Qualifier ::= NCName
```

936

937 See [section @Requires](#) for the formal definition of the @Requires annotation.

938 8.2 Specific Intent Annotations

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

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

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

```
950 @Integrity
```

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

```
952 @Authentication( {"message", "transport"} )
```

953 This annotation attaches the pair of qualified intents: "authentication.message" and
954 "authentication.transport" (the sca: namespace is assumed in this both of these cases –
955 "http://docs.oasis-open.org/ns/opencsa/sca/200912").

956 The general form of specific intent annotations is:

```
957 '@' Intent ('(' qualifiers ')')?
```

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

```
959 Intent      ::= NCName  
960 qualifiers  ::= "" qualifier "" (',' qualifier " ")*  
961 qualifier   ::= NCName ('.' qualifier)?  
962
```

963 8.2.1 How to Create Specific Intent Annotations

964 **SCA identifies annotations that correspond to intents by providing an @Intent annotation which**
965 **MUST be used in the definition of a specific intent annotation. [JCA70001]**

966 The @Intent annotation takes a single parameter, which (like the @Requires annotation) is the
967 String form of the QName of the intent. As part of the intent definition, it is good practice
968 (although not required) to also create String constants for the Namespace, for the Intent and for
969 Qualified versions of the Intent (if defined). These String constants are then available for use with
970 the @Requires annotation and it is also possible to use one or more of them as parameters to the
971 specific intent annotation.

972 Alternatively, the QName of the intent can be specified using separate parameters for the
973 targetNamespace and the localPart, for example:

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

975 See [section @Intent](#) for the formal definition of the @Intent annotation.

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

978 In this case, the attribute's definition needs to be marked with the @Qualifier annotation. The
979 @Qualifier tells SCA that the value of the attribute is treated as a qualifier for the intent
980 represented by the whole annotation. If more than one qualifier value is specified in an
981 annotation, it means that multiple qualified forms exist. For example:

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

983 implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport"
984 are set for the element to which the @Confidentiality annotation is attached.
985 See [section @Qualifier](#) for the formal definition of the @Qualifier annotation.
986 Examples of the use of the @Intent and the @Qualifier annotations in the definition of specific
987 intent annotations are shown in [the section dealing with Security Interaction Policy](#).

988 8.3 Application of Intent Annotations

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

- 990 • Java class
- 991 • Java interface
- 992 • Method
- 993 • Field
- 994 • Constructor parameter

995 **Intent annotations MUST NOT be applied to the following:**

- 996 • A method of a service implementation class, except for a setter method that is either
997 annotated with @Reference or introspected as an SCA reference according to the rules in
998 the appropriate Component Implementation specification
- 999 • A service implementation class field that is not either annotated with @Reference or
1000 introspected as an SCA reference according to the rules in the appropriate Component
1001 Implementation specification
- 1002 • A service implementation class constructor parameter that is not annotated with
1003 @Reference

1004 **[JCA70002]**

1005 Intent annotations can be applied to classes, interfaces, and interface methods. Applying an
1006 intent annotation to a field, setter method, or constructor parameter allows intents to be defined
1007 at references. Intent annotations can also be applied to reference interfaces and their methods.

1008 **Where multiple intent annotations (general or specific) are applied to the same Java element, the**
1009 **SCA runtime MUST compute the combined intents for the Java element by merging the intents**
1010 **from all intent annotations on the Java element according to the SCA Policy Framework [POLICY]**
1011 **rules for merging intents at the same hierarchy level. [JCA70003]**

1012 An example of multiple policy annotations being used together follows:

```
1013 @Authentication  
1014 @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

1015 In this case, the effective intents are "authentication", "confidentiality.message" and
1016 "integrity.message".

1017 **If intent annotations are specified on both an interface method and the method's declaring**
1018 **interface, the SCA runtime MUST compute the effective intents for the method by merging the**
1019 **combined intents from the method with the combined intents for the interface according to the**
1020 **SCA Policy Framework [POLICY] rules for merging intents within a structural hierarchy, with the**
1021 **method at the lower level and the interface at the higher level. [JCA70004]** This merging process
1022 does not remove or change any intents that are applied to the interface.

1023 8.3.1 Intent Annotation Examples

1024 The following examples show how the rules defined in section 8.3 are applied.

1025 Example 8.1 shows how intents on references are merged. In this example, the intents for myRef
1026 are "authentication" and "confidentiality.message".

```
1027 @Authentication
```

```

1028     @Requires(CONFIDENTIALITY)
1029     @Confidentiality("message")
1030     @Reference
1031     protected MyService myRef;
1032 
```

Example 8.1. Merging intents on references.

Example 8.2 shows that mutually exclusive intents cannot be applied to the same Java element. In this example, the Java code is in error because of contradictory mutually exclusive intents "managedTransaction" and "noManagedTransaction".

```

1036     @Requires({SCA_PREFIX+"managedTransaction",
1037              SCA_PREFIX+"noManagedTransaction"})
1038     @Reference
1039     protected MyService myRef;
1040 
```

Example 8.2. Mutually exclusive intents.

Example 8.3 shows that intents can be applied to Java service interfaces and their methods. In this example, the effective intents for `MyService.mymethod()` are "authentication" and "confidentiality".

```

1044     @Authentication
1045     public interface MyService {
1046         @Confidentiality
1047         public void mymethod();
1048     }
1049     @Service(MyService.class)
1050     public class MyServiceImpl {
1051         public void mymethod() {...}
1052     }
1053 
```

Example 8.3. Intents on Java interfaces, interface methods, and Java classes.

Example 8.4 shows that intents can be applied to Java service implementation classes. In this example, the effective intents for `MyService.mymethod()` are "authentication", "confidentiality", and "managedTransaction".

```

1057     @Authentication
1058     public interface MyService {
1059         @Confidentiality
1060         public void mymethod();
1061     }
1062     @Service(MyService.class)
1063     @Requires(SCA_PREFIX+"managedTransaction")
1064     public class MyServiceImpl {
1065         public void mymethod() {...}
1066     }
1067 
```

Example 8.4. Intents on Java service implementation classes.

Example 8.5 shows that intents can be applied to Java reference interfaces and their methods, and also to Java references. In this example, the effective intents for the method `mymethod()` of the reference `myRef` are "authentication", "integrity", and "confidentiality".

```

1071     @Authentication
1072     public interface MyRefInt {
1073         @Integrity
1074         public void mymethod();
1075     }
1076     @Service(MyService.class)
1077     public class MyServiceImpl {
1078         @Confidentiality
1079         @Reference
1080         protected MyRefInt myRef;
1081     }
1082 
```

1082 Example 8.5. Intents on Java references and their interfaces and methods.

1083 Example 8.6 shows that intents cannot be applied to methods of Java implementation classes. In
1084 this example, the Java code is in error because of the `@Authentication` intent annotation on the
1085 implementation method `MyServiceImpl.mymethod()`.

```
1086     public interface MyService {
1087         public void mymethod();
1088     }
1089     @Service(MyService.class)
1090     public class MyServiceImpl {
1091         @Authentication
1092         public void mymethod() {...}
1093     }
```

1094 Example 8.6. Intent on implementation method.

1095 Example 8.7 shows one effect of applying the SCA Policy Framework rules for merging intents
1096 within a structural hierarchy to Java service interfaces and their methods. In this example a
1097 qualified intent overrides an unqualified intent, so the effective intent for
1098 `MyService.mymethod()` is "confidentiality.message".

```
1099     @Confidentiality("message")
1100     public interface MyService {
1101         @Confidentiality
1102         public void mymethod();
1103     }
```

1104 Example 8.7. Merging qualified and unqualified intents on Java interfaces and methods.

1105 Example 8.8 shows another effect of applying the SCA Policy Framework rules for merging intents
1106 within a structural hierarchy to Java service interfaces and their methods. In this example a
1107 lower-level intent causes a mutually exclusive higher-level intent to be ignored, so the effective
1108 intent for `mymethod1()` is "managedTransaction" and the effective intent for `mymethod2()` is
1109 "noManagedTransaction".

```
1110     @Requires(SCA_PREFIX+"managedTransaction")
1111     public interface MyService {
1112         public void mymethod1();
1113         @Requires(SCA_PREFIX+"noManagedTransaction")
1114         public void mymethod2();
1115     }
```

1116 Example 8.8. Merging mutually exclusive intents on Java interfaces and methods.

1117 8.3.2 Inheritance and Annotation

1118 The following example shows the inheritance relations of intents on classes, operations, and super
1119 classes.

```
1120     package services.hello;
1121     import org.oasisopen.sca.annotation.Authentication;
1122     import org.oasisopen.sca.annotation.Integrity;
1123
1124     @Integrity("transport")
1125     @Authentication
1126     public class HelloService {
1127         @Integrity
1128         @Authentication("message")
1129         public String hello(String message) {...}
1130
1131         @Integrity
1132         @Authentication("transport")
1133         public String helloThere() {...}
1134     }
```

```

1135
1136 package services.hello;
1137 import org.oasisopen.sca.annotation.Authentication;
1138 import org.oasisopen.sca.annotation.Confidentiality;
1139
1140 @Confidentiality("message")
1141 public class HelloChildService extends HelloService {
1142     @Confidentiality("transport")
1143     public String hello(String message) {...}
1144     @Authentication
1145     String helloWorld() {...}
1146 }

```

1147 Example 8.9. Usage example of annotated policy and inheritance.

1148
1149 The effective intent annotation on the **helloWorld** method of **HelloChildService** is
1150 @Authentication and @Confidentiality("message").

1151 The effective intent annotation on the **hello** method of **HelloChildService** is
1152 @Confidentiality("transport"),

1153 The effective intent annotation on the **helloThere** method of **HelloChildService** is @Integrity
1154 and @Authentication("transport"), the same as for this method in the **HelloService** class.

1155 The effective intent annotation on the **hello** method of **HelloService** is @Integrity and
1156 @Authentication("message")

1157
1158 Table 8.1 below shows the equivalent declarative security interaction policy of the methods of the
1159 HelloService and HelloChildService implementations corresponding to the Java classes shown in
1160 Example 8.9.

	Method		
Class	hello()	helloThere()	helloWorld()
HelloService	integrity authentication.message	integrity authentication.transport	N/A
HelloChildService	confidentiality.transport	integrity authentication.transport	authentication confidentiality.message

1162
1163 Table 8.1. Declarative intents equivalent to annotated intents in Example 8.9.

1164 8.4 Relationship of Declarative and Annotated Intents

1165 Annotated intents on a Java class cannot be overridden by declarative intents in a composite
1166 document which uses the class as an implementation. This rule follows the general rule for intents
1167 that they represent requirements of an implementation in the form of a restriction that cannot be
1168 relaxed.

1169 However, a restriction can be made more restrictive so that an unqualified version of an intent
1170 expressed through an annotation in the Java class can be qualified by a declarative intent in a
1171 using composite document.

1172 8.5 Policy Set Annotations

1173 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies. For
1174 example, a concrete policy is the specific encryption algorithm to use when encrypting messages
1175 when using a specific communication protocol to link a reference to a service.

1176
1177 Policy Sets can be applied directly to Java implementations using the **@PolicySets** annotation.
1178 The @PolicySets annotation either takes the QName of a single policy set as a string or the name
1179 of two or more policy sets as an array of strings:

```
1180     '@PolicySets({' policySetQName (',' policySetQName )* '})'
```

1181

1182 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

1183 An example of the @PolicySets annotation:

1184

```
1185     @Reference(name="helloService", required=true)
1186     @PolicySets({ MY_NS + "WS_Encryption_Policy",
1187                 MY_NS + "WS_Authentication_Policy" })
1188     public setHelloService(HelloService service) {
1189         . . .
1190     }
1191
```

1192 In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both
1193 using the namespace defined for the constant MY_NS.

1194 PolicySets need to satisfy intents expressed for the implementation when both are present,
1195 according to the rules defined in [the Policy Framework specification \[POLICY\]](#).

1196 The SCA Policy Set annotation can be applied to the following Java elements:

- 1197 • Java class
- 1198 • Java interface
- 1199 • Method
- 1200 • Field
- 1201 • Constructor parameter

1202 **The @PolicySets annotation MUST NOT be applied to the following:**

- 1203 • A method of a service implementation class, except for a setter method that is either
1204 annotated with @Reference or introspected as an SCA reference according to the rules in
1205 the appropriate Component Implementation specification
- 1206 • A service implementation class field that is not either annotated with @Reference or
1207 introspected as an SCA reference according to the rules in the appropriate Component
1208 Implementation specification
- 1209 • A service implementation class constructor parameter that is not annotated with
1210 @Reference

1211 **[JCA70005]**

1212 The @PolicySets annotation can be applied to classes, interfaces, and interface methods. Applying
1213 a @PolicySets annotation to a field, setter method, or constructor parameter allows policy sets to
1214 be defined at references. The @PolicySets annotation can also be applied to reference interfaces
1215 and their methods.

1216 **If the @PolicySets annotation is specified on both an interface method and the method's declaring**
1217 **interface, the SCA runtime MUST compute the effective policy sets for the method by merging the**
1218 **policy sets from the method with the policy sets from the interface. [JCA70006]** This merging
1219 process does not remove or change any policy sets that are applied to the interface.

1220 8.6 Security Policy Annotations

1221 This section introduces annotations for commonly used SCA security intents, as defined in [the SCA](#)
1222 [Policy Framework Specification \[POLICY\]](#). Also see the SCA Policy Framework Specification for
1223 additional security policy intents that can be used with the @Requires annotation. The following
1224 annotations for security policy intents and qualifiers are defined:

- 1225 • @Authentication
- 1226 • @Authorization
- 1227 • @Confidentiality
- 1228 • @Integrity
- 1229 • @MutualAuthentication

1230 The @Authentication, @Confidentiality, and @Integrity intents have the same pair of Qualifiers:

- 1231 • message
- 1232 • transport

1233 The formal definitions of the security intent annotations are found in the section "Java
1234 Annotations".

1235 The following example shows an example of applying security intents to the setter method used to
1236 inject a reference. Accessing the hello operation of the referenced HelloService requires both
1237 "integrity.message" and "authentication.message" intents to be honored.

```
1238
1239 package services.hello;
1240 // Interface for HelloService
1241 public interface HelloService {
1242     String hello(String helloMsg);
1243 }
1244
1245 package services.client;
1246 // Interface for ClientService
1247 public interface ClientService {
1248     public void clientMethod();
1249 }
1250
1251 // Implementation class for ClientService
1252 package services.client;
1253
1254 import services.hello.HelloService;
1255 import org.oasisopen.sca.annotation.*;
1256
1257 @Service(ClientService.class)
1258 public class ClientServiceImpl implements ClientService {
1259
1260     private HelloService helloService;
1261
1262     @Reference(name="helloService", required=true)
1263     @Integrity("message")
1264     @Authentication("message")
1265     public void setHelloService(HelloService service) {
1266         helloService = service;
1267     }
1268
1269     public void clientMethod() {
1270         String result = helloService.hello("Hello World!");

```

```
1271         ...
1272     }
1273 }
```

1274

1275 Example 8.10. Usage of security intents on a reference.

1276

1277 8.7 Transaction Policy Annotations

1278 This section introduces annotations for commonly used SCA transaction intents, as defined in [the](#)
1279 [SCA Policy Framework specification \[POLICY\]](#). Also see the SCA Policy Framework Specification for
1280 additional transaction policy intents that can be used with the @Requires annotation. The following
1281 annotations for transaction policy intents and qualifiers are defined:

- 1282 • @ManagedTransaction
- 1283 • @NoManagedTransaction
- 1284 • @SharedManagedTransaction

1285

1286 The @ManagedTransaction intent has the following Qualifiers:

- 1287 • global
- 1288 • local

1289

1290 The formal definitions of the transaction intent annotations are found in the section “Java
1291 Annotations”.

1292 The following example shows an example of applying a transaction intent to a component
1293 implementation, where the component implementation requires a global transaction.

1294

```
1295 package services.hello;
1296 // Interface for HelloService
1297 public interface HelloService {
1298     String hello(String helloMsg);
1299 }
1300
1301 // Implementation class for HelloService
1302 package services.hello.impl;
1303
1304 import services.hello.HelloService;
1305 import org.oasisopen.sca.annotation.*;
1306
1307 @Service(HelloService.class)
1308 @ManagedTransaction("global")
1309 public class HelloServiceImpl implements HelloService {
1310
1311     public void someMethod() {
1312         ...
1313     }
1314 }
```

1315

1316 Example 8.11. Usage of transaction intents in an implementation.

1317

9 Java API

1318

1319 This section provides a reference for the Java API offered by SCA.

9.1 Component Context

1320

1321 The following Java code defines the **ComponentContext** interface:

1322

```
1323 package org.oasisopen.sca;
1324 import java.util.Collection;
1325 public interface ComponentContext {
1326
1327     String getURI();
1328
1329     <B> B getService(Class<B> businessInterface, String referenceName);
1330
1331     <B> ServiceReference<B> getServiceReference( Class<B> businessInterface,
1332                                               String referenceName);
1333     <B> Collection<B> getServices( Class<B> businessInterface,
1334                                 String referenceName);
1335
1336     <B> Collection<ServiceReference<B>> getServiceReferences(
1337                                     Class<B> businessInterface,
1338                                     String referenceName);
1339
1340     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface);
1341
1342     <B> ServiceReference<B> createSelfReference( Class<B> businessInterface,
1343                                               String serviceName);
1344
1345     <B> B getProperty(Class<B> type, String propertyName);
1346
1347     RequestContext getRequestContext();
1348
1349     <B> ServiceReference<B> cast(B target) throws IllegalArgumentException;
1350
1351 }
```

1352 *Figure 9-1: ComponentContext interface*

getURI () method:

1353

1354 Returns the absolute URI of the component within the SCA Domain.

1355 Returns:

- 1356 • **String** which contains the absolute URI of the component in the SCA Domain
- 1357 The `ComponentContext.getURI` method **MUST** return the absolute URI of the component in
- 1358 the SCA Domain. [\[JCA80008\]](#)

1359 Parameters:

- 1360 • **none**

1361 Exceptions:

- 1362 • **none**

1363

getService (Class businessInterface, String referenceName) method:

1364

1365 Returns a typed service proxy object for a reference defined by the current component, where the
1366 reference has multiplicity 0..1 or 1..1.

1367 Returns:

- 1368 • **B** which is a proxy object for the reference, which implements the interface B contained in
1369 the businessInterface parameter.
1370 The ComponentContext.getService method MUST return the proxy object implementing
1371 the interface provided by the businessInterface parameter, for the reference named by the
1372 referenceName parameter with the interface defined by the businessInterface parameter
1373 when that reference has a target service configured. [JCA80009]
1374 The ComponentContext.getService method MUST return null if the multiplicity of the
1375 reference named by the referenceName parameter is 0..1 and the reference has no target
1376 service configured. [JCA80010]

1377 Parameters:

- 1378 • **Class businessInterface** - the Java interface for the service reference
- 1379 • **String referenceName** - the name of the service reference

1380 Exceptions:

- 1381 • The ComponentContext.getService method MUST throw an IllegalArgumentException if the
1382 reference identified by the referenceName parameter has multiplicity of 0..n or 1..n.
1383 [JCA80001]
- 1384 • The ComponentContext.getService method MUST throw an IllegalArgumentException if the
1385 component does not have a reference with the name supplied in the referenceName
1386 parameter. [JCA80011]
- 1387 • The ComponentContext.getService method MUST throw an IllegalArgumentException if the
1388 service reference with the name supplied in the referenceName does not have an interface
1389 compatible with the interface supplied in the businessInterface parameter. [JCA80012]

1390

1391 **getServiceReference (Class businessInterface, String referenceName) method:**

1392 Returns a ServiceReference object for a reference defined by the current component, where the
1393 reference has multiplicity 0..1 or 1..1.

1394 Returns:

- 1395 • **ServiceReference** which is a ServiceReference proxy object for the reference, which
1396 implements the interface contained in the businessInterface parameter.
1397 The ComponentContext.getServiceReference method MUST return a ServiceReference
1398 object typed by the interface provided by the businessInterface parameter, for the
1399 reference named by the referenceName parameter with the interface defined by the
1400 businessInterface parameter when that reference has a target service configured.
1401 [JCA80013]
1402 The ComponentContext.getServiceReference method MUST return null if the multiplicity of
1403 the reference named by the referenceName parameter is 0..1 and the reference has no
1404 target service configured. [JCA80007]

1405 Parameters:

- 1406 • **Class businessInterface** - the Java interface for the service reference
- 1407 • **String referenceName** - the name of the service reference

1408 Exceptions:

- 1409 The ComponentContext.getServiceReference method MUST throw an
1410 IllegalArgumentException if the reference named by the referenceName parameter has
1411 multiplicity greater than one. [JCA80004]
1412 The ComponentContext.getServiceReference method MUST throw an
1413 IllegalArgumentException if the reference named by the referenceName parameter does
1414 not have an interface of the type defined by the businessInterface parameter. [JCA80005]

1415 The ComponentContext.getServiceReference method MUST throw an
1416 IllegalArgumentException if the component does not have a reference with the name
1417 provided in the referenceName parameter. [JCA80006]

1418

1419 **getServices(Class businessInterface, String referenceName) method:**

1420 Returns a list of typed service proxies for a reference defined by the current component, where
1421 the reference has multiplicity 0..n or 1..n.

1422 Returns:

- 1423 • **Collection** which is a collection of proxy objects for the reference, one for each
1424 target service to which the reference is wired, where each proxy object implements the
1425 interface B contained in the businessInterface parameter.
1426 The ComponentContext.getServices method MUST return a collection containing one proxy
1427 object implementing the interface provided by the businessInterface parameter for each of
1428 the target services configured on the reference identified by the referenceName
1429 parameter. [JCA80014]
1430 The ComponentContext.getServices method MUST return an empty collection if the service
1431 reference with the name supplied in the referenceName parameter is not wired to any
1432 target services. [JCA80015]

1433 Parameters:

- 1434 • **Class businessInterface** - the Java interface for the service reference
- 1435 • **String referenceName** - the name of the service reference

1436 Exceptions:

- 1437 • The ComponentContext.getServices method MUST throw an IllegalArgumentException if
1438 the reference identified by the referenceName parameter has multiplicity of 0..1 or 1..1.
1439 [JCA80016]
- 1440 • The ComponentContext.getServices method MUST throw an IllegalArgumentException if
1441 the component does not have a reference with the name supplied in the referenceName
1442 parameter. [JCA80017]
- 1443 • The ComponentContext.getServices method MUST throw an IllegalArgumentException if
1444 the service reference with the name supplied in the referenceName does not have an
1445 interface compatible with the interface supplied in the businessInterface
1446 parameter. [JCA80018]

1447

1448 **getServiceReferences(Class businessInterface, String referenceName) method:**

1449 Returns a list of typed ServiceReference objects for a reference defined by the current component,
1450 where the reference has multiplicity 0..n or 1..n.

1451 Returns:

- 1452 • **Collection<ServiceReference>** which is a collection of ServiceReference objects for
1453 the reference, one for each target service to which the reference is wired, where each
1454 proxy object implements the interface B contained in the businessInterface parameter.
1455 The collection is empty if the reference is not wired to any target services.
1456 The ComponentContext.getServiceReferences method MUST return a collection containing
1457 one ServiceReference object typed by the interface provided by the businessInterface
1458 parameter for each of the target services configured on the reference identified by the
1459 referenceName parameter. [JCA80019]
1460 The ComponentContext.getServiceReferences method MUST return an empty collection if
1461 the service reference with the name supplied in the referenceName parameter is not wired
1462 to any target services. [JCA80020]

1463 •

1464 Parameters:

- 1465 • **Class businessInterface** - the Java interface for the service reference
- 1466 • **String referenceName** - the name of the service reference

1467 Exceptions:

- 1468 • The `ComponentContext.getServiceReferences` method MUST throw an
1469 `IllegalArgumentException` if the reference identified by the `referenceName` parameter has
1470 multiplicity of 0..1 or 1..1. [JCA80021]
- 1471 • The `ComponentContext.getServiceReferences` method MUST throw an
1472 `IllegalArgumentException` if the component does not have a reference with the name
1473 supplied in the `referenceName` parameter. [JCA80022]
- 1474 • The `ComponentContext.getServiceReferences` method MUST throw an
1475 `IllegalArgumentException` if the service reference with the name supplied in the
1476 `referenceName` does not have an interface compatible with the interface supplied in the
1477 `businessInterface` parameter. [JCA80023]

1478

1479 ***createSelfReference(Class businessInterface) method:***

1480 Returns a `ServiceReference` object that can be used to invoke this component over the designated
1481 service.

1482 Returns:

- 1483 • **ServiceReference** which is a `ServiceReference` object for the service of this
1484 component which has the supplied business interface. If the component has multiple
1485 services with the same business interface the SCA runtime can return a `ServiceReference`
1486 for any one of them.
1487 The `ComponentContext.createSelfReference` method MUST return a `ServiceReference`
1488 object typed by the interface defined by the `businessInterface` parameter for one of the
1489 services of the invoking component which has the interface defined by the
1490 `businessInterface` parameter. [JCA80024]

1491 Parameters:

- 1492 • **Class businessInterface** - the Java interface for the service

1493 Exceptions:

- 1494 • The `ComponentContext.getServiceReferences` method MUST throw an
1495 `IllegalArgumentException` if the component does not have a service which implements the
1496 interface identified by the `businessInterface` parameter. [JCA80025]

1497

1498 ***createSelfReference(Class businessInterface, String serviceName) method:***

1499 Returns a `ServiceReference` that can be used to invoke this component over the designated
1500 service. The `serviceName` parameter explicitly declares the service name to invoke

1501 Returns:

- 1502 • **ServiceReference** which is a `ServiceReference` proxy object for the reference, which
1503 implements the interface contained in the `businessInterface` parameter.
1504 The `ComponentContext.createSelfReference` method MUST return a `ServiceReference`
1505 object typed by the interface defined by the `businessInterface` parameter for the service
1506 identified by the `serviceName` of the invoking component and which has the interface
1507 defined by the `businessInterface` parameter. [JCA80026]

1508 Parameters:

- 1509 • **Class businessInterface** - the Java interface for the service reference
- 1510 • **String serviceName** - the name of the service reference

1511 Exceptions:

- 1512
- The `ComponentContext.createSelfReference` method MUST throw an `IllegalArgumentException` if the component does not have a service with the name identified by the `serviceName` parameter. [JCA80027]
- 1513
- The `ComponentContext.createSelfReference` method MUST throw an `IllegalArgumentException` if the component service with the name identified by the `serviceName` parameter does not implement a business interface which is compatible with the supplied `businessInterface` parameter. [JCA80028]
- 1514
- 1515
- 1516
- 1517
- 1518

1519

1520 ***getProperty (Class type, String propertyName) method:***

1521 Returns the value of an SCA property defined by this component.

1522 Returns:

- **** which is an object of the type identified by the type parameter containing the value specified for the property in the SCA configuration of the component. *null* if the SCA configuration of the component does not specify any value for the property. The `ComponentContext.getProperty` method MUST return an object of the type identified by the type parameter containing the value specified in the component configuration for the property named by the `propertyName` parameter or null if no value is specified in the configuration. [JCA80029]

1523

1524

1525

1526

1527

1528

1529

1530 Parameters:

- **Class type** - the Java class of the property (Object mapped type for primitive Java types - e.g. Integer if the type is int)
- **String propertyName** - the name of the property

1531

1532

1533

1534 Exceptions:

- The `ComponentContext.getProperty` method MUST throw an `IllegalArgumentException` if the component does not have a property with the name identified by the `propertyName` parameter. [JCA80030]
- The `ComponentContext.getProperty` method MUST throw an `IllegalArgumentException` if the component property with the name identified by the `propertyName` parameter does not have a type which is compatible with the supplied type parameter. [JCA80031]

1535

1536

1537

1538

1539

1540

1541

1542 ***getRequestContext() method:***

1543 Returns the `RequestContext` for the current SCA service request.

1544 Returns:

- **RequestContext** which is the `RequestContext` object for the current SCA service invocation. *null* if there is no current request or if the context is unavailable. The `ComponentContext.getRequestContext` method MUST return a non-null `RequestContext` object when invoked during the execution of a Java business method for a service operation or a callback operation on the same thread that the SCA runtime provided, and MUST return null in all other cases. [JCA80002]

1545

1546

1547

1548

1549

1550

1551

1552 Parameters:

- *none*

1553

1554 Exceptions:

- *none*

1555

1556

1557 ***cast(B target) method:***

1558 Casts a type-safe reference to a `ServiceReference`

1559 Returns:

- 1560 • **ServiceReference** which is a ServiceReference object which implements the same
1561 business interface B as a reference proxy object
1562 The ComponentContext.cast method MUST return a ServiceReference object which is
1563 typed by the same business interface as specified by the reference proxy object supplied
1564 in the target parameter. [JCA80032]
1565

1566 Parameters:

- 1567 • **B target** - a type safe reference proxy object which implements the business interface B

1568 Exceptions:

- 1569 • The ComponentContext.cast method MUST return a ServiceReference object which is
1570 typed by the same business interface as specified by the reference proxy object supplied
1571 in the target parameter. [JCA80033]

1572

1573 A component can access its component context by defining a field or setter method typed by
1574 **org.oasisopen.sca.ComponentContext** and annotated with **@Context**. To access a target
1575 service, the component uses **ComponentContext.getService(..)**.

1576 Figure 9-2 shows an example of component context usage in a Java class using the @Context
1577 annotation.

```
1578 private ComponentContext componentContext;  
1579  
1580 @Context  
1581 public void setContext(ComponentContext context) {  
1582     componentContext = context;  
1583 }  
1584  
1585 public void doSomething() {  
1586     HelloWorld service =  
1587     componentContext.getService(HelloWorld.class, "HelloWorldComponent");  
1588     service.hello("hello");  
1589 }
```

1590 *Figure 9-2: ComponentContext injection example*

1591 Similarly, non-SCA client code can use the ComponentContext API to perform operations against a
1592 component in an SCA domain. How the non-SCA client code obtains a reference to a
1593 ComponentContext is runtime specific.

1594 9.2 Request Context

1595 Figure 9-3 shows the **RequestContext** interface:
1596

```
1597 package org.oasisopen.sca;  
1598  
1599 import javax.security.auth.Subject;  
1600  
1601 public interface RequestContext {  
1602  
1603     Subject getSecuritySubject();  
1604  
1605     String getServiceName();  
1606     <CB> ServiceReference<CB> getCallbackReference();  
1607     <CB> CB getCallback();  
1608     <B> ServiceReference<B> getServiceReference();
```

1609 }

1610 *Figure 9-3: RequestContext interface*

1611

1612 **getSecuritySubject () method:**

1613 Returns the JAAS Subject of the current request (see [the JAAS Reference Guide \[JAAS\]](#) for details

1614 of JAAS).

1615 Returns:

- 1616 • **javax.security.auth.Subject** object which is the JAAS subject for the request.
- 1617 null if there is no subject for the request.
- 1618 The RequestContext.getSecuritySubject method MUST return the JAAS subject of the
- 1619 current request, or null if there is no subject or null if the method is invoked from code not
- 1620 processing a service request or callback request. [JCA80034]

1621 Parameters:

- 1622 • **none**

1623 Exceptions:

- 1624 • **none**

1625

1626 **getServiceName () method:**

1627 Returns the name of the service on the Java implementation the request came in on.

1628 Returns:

- 1629 • **String** containing the name of the service. **null** if the method is invoked from a thread
- 1630 that is not processing a service operation or a callback operation.
- 1631 The RequestContext.getServiceName method MUST return the name of the service for
- 1632 which an operation is being processed, or null if invoked from a thread that is not
- 1633 processing a service operation or a callback operation. [JCA80035]

1634 Parameters:

- 1635 • **none**

1636 Exceptions:

- 1637 • **none**

1638

1639 **getCallbackReference () method:**

1640 Returns a service reference proxy for the callback for the invoked service operation, as specified

1641 by the service client.

1642 Returns:

- 1643 • **ServiceReference<CB>** which is a service reference for the callback for the invoked
- 1644 service, as supplied by the service client. It is typed with the callback interface.
- 1645 **null** if the invoked service has an interface which is not bidirectional or if the
- 1646 getCallbackReference() method is called during the processing of a callback operation.
- 1647 **null** if the method is invoked from a thread that is not processing a service operation.
- 1648 The RequestContext.getCallbackReference method MUST return a ServiceReference object
- 1649 typed by the interface of the callback supplied by the client of the invoked service, or null
- 1650 if either the invoked service is not bidirectional or if the method is invoked from a thread
- 1651 that is not processing a service operation. [JCA80036]

1652 Parameters:

- 1653 • **none**

1654 Exceptions:

- 1655 • **none**

1656

1657 **getCallback () method:**

1658 Returns a proxy for the callback for the invoked service as specified by the service client.

1659 Returns:

- 1660 • **CB** proxy object for the callback for the invoked service as supplied by the service client.
1661 It is typed with the callback interface.
1662 **null** if the invoked service has an interface which is not bidirectional or if the getCallback()
1663 method is called during the processing of a callback operation.
1664 **null** if the method is invoked from a thread that is not processing a service operation.
1665 The RequestContext.getCallback method MUST return a reference proxy object typed by
1666 the interface of the callback supplied by the client of the invoked service, or null if either
1667 the invoked service is not bidirectional or if the method is invoked from a thread that is
1668 not processing a service operation. [JCA80037]

1669 Parameters:

- 1670 • **none**

1671 Exceptions:

- 1672 • **none**

1673

1674 **getServiceReference () method:**

1675 Returns a ServiceReference object for the service that was invoked.

1676

1677 Returns:

- 1678 • **ServiceReference** which is a service reference for the invoked service. It is typed
1679 with the interface of the service.
1680 Returns **null** if the method is invoked from a thread that is not processing a service
1681 operation or a callback operation.

1682
1683 When invoked during the execution of a service operation, the
1684 RequestContext.getServiceReference method MUST return a ServiceReference that
1685 represents the service that was invoked. [JCA80003]

1686 When invoked during the execution of a callback operation, the
1687 RequestContext.getServiceReference method MUST return a ServiceReference that
1688 represents the callback that was invoked. [JCA80038]

1689 When invoked from a thread not involved in the execution of either a service operation or
1690 of a callback operation, the RequestContext.getServiceReference method MUST return
1691 null. [JCA80039]

1692 Parameters:

- 1693 • **none**

1694 Exceptions:

- 1695 • **none**

1696

1697 ServiceReferences can be injected using the @Reference annotation on a field, a setter method, or
1698 constructor parameter taking the type ServiceReference. The detailed description of the usage of
1699 these methods is described in the section on Asynchronous Programming in this document.

1700

1701 9.3 ServiceReference Interface

1702 ServiceReferences can be injected using the @Reference annotation on a field, a setter method, or
1703 constructor parameter taking the type ServiceReference. The detailed description of the usage of
1704 these methods is described in the section on Asynchronous Programming in this document.

1705 Figure 9-4 defines the **ServiceReference** interface:

```
1706 package org.oasisopen.sca;  
1707  
1708 public interface ServiceReference<B> extends java.io.Serializable {  
1709  
1710     B getService();  
1711     Class<B> getBusinessInterface();  
1712 }  
1713
```

1714 *Figure 9-4: RequestContext interface*

1715

1716 The ServiceReference interface has the following methods:

1717 **getService () method:**

1718 Returns a type-safe reference to the target of this reference. The instance returned is guaranteed
1719 to implement the business interface for this reference. The value returned is a proxy to the target
1720 that implements the business interface associated with this reference.

1721 Returns:

- 1722 • **** which is type-safe reference proxy object to the target of this reference. It is typed
1723 with the interface of the target service.
1724 The ServiceReference.getService method MUST return a reference proxy object which can
1725 be used to invoke operations on the target service of the reference and which is typed
1726 with the business interface of the reference. [JCA80040]
1727

1728 Parameters:

- 1729 • **none**

1730 Exceptions:

- 1731 • **none**

1732

1733 **getBusinessInterface () method:**

1734 Returns the Java class for the business interface associated with this ServiceReference.

1735 Returns:

- 1736 • **Class** which is a Class object of the business interface associated with the reference.
1737 The ServiceReference.getBusinessInterface method MUST return a Class object
1738 representing the business interface of the reference. [JCA80041]
1739

1740 Parameters:

- 1741 • **none**

1742 Exceptions:

- 1743 • **none**

1744

1745 9.4 ResponseDispatch interface

1746 The **ResponseDispatch** interface is shown in Figure 9-5:

```
1747 package org.oasisopen.sca;  
1748  
1749 public interface ResponseDispatch<T> {  
1750     void sendResponse(T res);  
1751     void sendFault(Throwable e);  
1752     Map<String, Object> getContext();  
1753 }
```

1754 *Figure 9-5: ResponseDispatch interface*

1755 **sendResponse (T response) method:**

1756 Sends the response message from an asynchronous service method. This method can only be
1757 invoked once for a given ResponseDispatch object and cannot be invoked if sendFault has
1758 previously been invoked for the same ResponseDispatch object.

1759 Returns:

- 1760 • **void**
1761 The ResponseDispatch.sendResponse() method MUST send the response message to the
1762 client of an asynchronous service. [JCA50057]

1763 Parameters:

- 1764 • **T** - an instance of the response message returned by the service operation

1765 Exceptions:

- 1766 • The ResponseDispatch.sendResponse() method MUST throw an IllegalStateException if
1767 either the sendResponse method or the sendFault method has already been called once.
1768 [JCA80058]

1769

1770 **sendFault (Throwable e) method:**

1771 Sends an exception as a fault from an asynchronous service method. This method can only be
1772 invoked once for a given ResponseDispatch object and cannot be invoked if sendResponse has
1773 previously been invoked for the same ResponseDispatch object.

1774 Returns:

- 1775 • **void**
1776 The ResponseDispatch.sendFault() method MUST send the supplied fault to the client of an
1777 asynchronous service. [JCA80059]

1778 Parameters:

- 1779 • **e** - an instance of an exception returned by the service operation

1780 Exceptions:

- 1781 • The ResponseDispatch.sendFault() method MUST throw an IllegalStateException if either
1782 the sendResponse method or the sendFault method has already been called once.
1783 [JCA50060]

1784

1785 **getContext () method:**

1786 Obtains the context object for the ResponseDispatch method

1787 Returns:

- 1788 • **Map<String, object>** which is the context object for the ResponseDispatch object.
1789 The invoker can update the context object with appropriate context information, prior to
1790 invoking either the sendResponse method or the sendFault method

1791 Parameters:

- 1792 • **none**

1793 Exceptions:

- 1794 • **none**

1795

1796 **9.5 ServiceRuntimeException**

1797 Figure 9-6 shows the **ServiceRuntimeException**.

1798

```
1799   package org.oasisopen.sca;  
1800  
1801   public class ServiceRuntimeException extends RuntimeException {  
1802       ...  
1803   }
```

1804 Figure 9-6: ServiceRuntimeException

1805

1806 This exception signals problems in the management of SCA component execution.

1807 **9.6 ServiceUnavailableException**

1808 Figure 9-7 shows the **ServiceUnavailableException**.

1809

```
1810   package org.oasisopen.sca;  
1811  
1812   public class ServiceUnavailableException extends ServiceRuntimeException {  
1813       ...  
1814   }
```

1815 Figure 9-7: ServiceUnavailableException

1816 This exception signals problems in the interaction with remote services. These are exceptions
1817 that can be transient, so retrying is appropriate. Any exception that is a ServiceRuntimeException
1818 that is *not* a ServiceUnavailableException is unlikely to be resolved by retrying the operation, since
1819 it most likely requires human intervention

1820 **9.7 InvalidServiceException**

1821 Figure 9-8 shows the **InvalidServiceException**.

1822

```
1823   package org.oasisopen.sca;  
1824  
1825   public class InvalidServiceException extends ServiceRuntimeException {  
1826       ...  
1827   }
```

1828 Figure 9-8: InvalidServiceException

1829

1830 This exception signals that the ServiceReference is no longer valid. This can happen when the
1831 target of the reference is undeployed. This exception is not transient and therefore is unlikely to
1832 be resolved by retrying the operation and will most likely require human intervention.

1833 9.8 Constants

1834 The SCA **Constants** interface defines a number of constant values that are used in the SCA Java
1835 APIs and Annotations. Figure 9-9 shows the Constants interface:

```
1836 package org.oasisopen.sca;  
1837  
1838 public interface Constants {  
1839     String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200912";  
1840     String SCA_PREFIX = "{"+SCA_NS+"}";  
1841 }
```

1842 *Figure 9-9: Constants interface*

1843

1844 9.9 SCAClientFactory Class

1845 The SCAClientFactory class provides the means for client code to obtain a proxy reference object
1846 for a service within an SCA Domain, through which the client code can invoke operations of that
1847 service. This is particularly useful for client code that is running outside the SCA Domain
1848 containing the target service, for example where the code is "unmanaged" and is not running
1849 under an SCA runtime.

1850 The SCAClientFactory is an abstract class which provides a set of static newInstance(...) methods
1851 which the client can invoke in order to obtain a concrete object implementing the
1852 SCAClientFactory interface for a particular SCA Domain. The returned SCAClientFactory object
1853 provides a getService() method which provides the client with the means to obtain a reference
1854 proxy object for a service running in the SCA Domain.

1855 The SCAClientFactory class is shown in Figure 9-10:

1856

```
1857 /*  
1858  * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.  
1859  * OASIS trademark, IPR and other policies apply.  
1860  */  
1861 package org.oasisopen.sca.client;  
1862  
1863 import java.net.URI;  
1864 import java.util.Properties;  
1865  
1866 import org.oasisopen.sca.NoSuchDomainException;  
1867 import org.oasisopen.sca.NoSuchServiceException;  
1868 import org.oasisopen.sca.client.SCAClientFactoryFinder;  
1869 import org.oasisopen.sca.client.impl.SCAClientFactoryFinderImpl;  
1870  
1871 /**  
1872  * The SCAClientFactory can be used by non-SCA managed code to  
1873  * lookup services that exist in a SCADomain.  
1874  *  
1875  * @see SCAClientFactoryFinderImpl  
1876  *  
1877  * @author OASIS Open  
1878  */  
1879 public abstract class SCAClientFactory {  
1880
```

```

1881
1882     protected static SCAClientFactoryFinder factoryFinder;
1883
1884     private SCAClientFactory() {}
1885
1886     protected SCAClientFactory(URI domainURI)
1887         throws NoSuchDomainException {...}
1888
1889     public URI getDomainURI() {...}
1890
1891     public static SCAClientFactory newInstance( URI domainURI )
1892         throws NoSuchDomainException {...}
1893
1894     public static SCAClientFactory newInstance(Properties properties,
1895                                             URI domainURI)
1896         throws NoSuchDomainException {...}
1897
1898     public static SCAClientFactory newInstance(ClassLoader classLoader,
1899                                             URI domainURI)
1900         throws NoSuchDomainException {...}
1901
1902     public static SCAClientFactory newInstance(Properties properties,
1903                                             ClassLoader classLoader,
1904                                             URI domainURI)
1905         throws NoSuchDomainException {...}
1906
1907     public abstract <T> T getService(Class<T> interfaze, String serviceURI)
1908         throws NoSuchServiceException, NoSuchDomainException;
1909 }

```

Figure 9-10: SCAClientFactory class

newInstance (URI domainURI) method:

Obtains a object implementing the SCAClientFactory class.

Returns:

- **object** which implements the SCAClientFactory class
The SCAClientFactory.newInstance(URI) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter. [JCA80042]

Parameters:

- **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

Exceptions:

- The SCAClientFactory.newInstance(URI) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain. [JCA80043]

newInstance(Properties properties, URI domainURI) method:

Obtains a object implementing the SCAClientFactory class, using a specified set of properties.

Returns:

- **object** which implements the SCAClientFactory class
The SCAClientFactory.newInstance(Properties, URI) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter. [JCA80044]

1932 Parameters:

1933 • **properties** - a set of Properties that can be used when creating the object which
1934 implements the SCAClientFactory class.

1935 • **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient
1936 object

1937 Exceptions:

1938 • The SCAClientFactory.newInstance(Properties, URI) method MUST throw a
1939 NoSuchDomainException if the domainURI parameter does not identify a valid SCA
1940 Domain. [JCA80045]

1941 **newInstance(Classloader classLoader, URI domainURI) method:**

1942 Obtains a object implementing the SCAClientFactory class using a specified classloader.

1943 Returns:

1944 • **object** which implements the SCAClientFactory class
1945 The SCAClientFactory.newInstance(Classloader, URI) method MUST return an object
1946 which implements the SCAClientFactory class for the SCA Domain identified by the
1947 domainURI parameter. [JCA80046]

1948 Parameters:

1949 • **classLoader** - a ClassLoader to use when creating the object which implements the
1950 SCAClientFactory class.

1951 • **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient
1952 object

1953 Exceptions:

1954 • The SCAClientFactory.newInstance(Classloader, URI) method MUST throw a
1955 NoSuchDomainException if the domainURI parameter does not identify a valid SCA
1956 Domain. [JCA80047]

1957 **newInstance(Properties properties, Classloader classLoader, URI domainURI) method:**

1958 Obtains a object implementing the SCAClientFactory class using a specified set of properties and a
1959 specified classloader.

1960 Returns:

1961 • **object** which implements the SCAClientFactory class
1962 The SCAClientFactory.newInstance(Properties, Classloader, URI) method MUST return an
1963 object which implements the SCAClientFactory class for the SCA Domain identified by the
1964 domainURI parameter. [JCA80048]

1965 Parameters:

1966 • **properties** - a set of Properties that can be used when creating the object which
1967 implements the SCAClientFactory class.

1968 • **classLoader** - a ClassLoader to use when creating the object which implements the
1969 SCAClientFactory class.

1970 • **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient
1971 object

1972 Exceptions:

1973 • The SCAClientFactory.newInstance(Properties, Classloader, URI) MUST throw a
1974 NoSuchDomainException if the domainURI parameter does not identify a valid SCA
1975 Domain. [JCA80049]

1976

1977 **getService(Class<T> interfaze, String serviceURI) method:**

1978 Obtains a proxy reference object for a specified target service in a specified SCA Domain.

1979 Returns:

1980 • **<T>** a proxy object which implements the business interface T

1981 Invocations of a business method of the proxy causes the invocation of the corresponding

1982 operation of the target service.

1983 The SCAClientFactory.getService method MUST return a proxy object which implements

1984 the business interface defined by the interfaze parameter and which can be used to invoke

1985 operations on the service identified by the serviceURI parameter. [JCA80050]

1986 Parameters:

1987 • **interfaze** - a Java interface class which is the business interface of the target service

1988 • **serviceURI** - a String containing the relative URI of the target service within its SCA

1989 Domain.

1990 Takes the form componentName/serviceName or can also take the extended form

1991 componentName/serviceName/bindingName to use a specific binding of the target service

1992 Exceptions:

1993 • The SCAClientFactory.getService method MUST throw a NoSuchServiceException if a

1994 service with the relative URI serviceURI and a business interface which matches interfaze

1995 cannot be found in the SCA Domain targeted by the SCAClient object. [JCA80051]

1996 • The SCAClientFactory.getService method MUST throw a NoSuchServiceException if the

1997 domainURI of the SCAClientFactory does not identify a valid SCA Domain. [JCA80052]

1998

1999 **SCAClientFactory (URI) method:** a single argument constructor that must be available on all

2000 concrete subclasses of SCAClientFactory. The URI required is the URI of the Domain targeted by

2001 the SCAClientFactory

2002 **getDomainURI() method:**

2003 Obtains the Domain URI value for this SCAClientFactory

2004 Returns:

2005 • **URI** of the target SCA Domain for this SCAClientFactory

2006 The SCAClientFactory.getDomainURI method MUST return the SCA Domain URI of the

2007 Domain associated with the SCAClientFactory object. [JCA80053]

2008 Parameters:

2009 • **none**

2010 Exceptions:

2011 • The SCAClientFactory.getDomainURI method MUST throw a **NoSuchServiceException** if

2012 the domainURI of the SCAClientFactory does not identify a valid SCA Domain. [JCA80054]

2013

2014 **private SCAClientFactory() method:**

2015 This private no-argument constructor prevents instantiation of an SCAClientFactory instance

2016 without the use of the constructor with an argument, even by subclasses of the abstract

2017 SCAClientFactory class.

2018 **factoryFinder protected field:**

2019 Provides a means by which a provider of an SCAClientFactory implementation can inject a factory

2020 finder implementation into the abstract SCAClientFactory class - once this is done, future

2021 invocations of the SCAClientFactory use the injected factory finder to locate and return an instance

2022 of a subclass of SCAClientFactory.

2023

2024 9.10 SCAClientFactoryFinder Interface

2025 The SCAClientFactoryFinder interface is a Service Provider Interface representing a
2026 SCAClientFactory finder. SCA provides a default reference implementation of this interface. SCA
2027 runtime vendors can create alternative implementations of this interface that use different class
2028 loading or lookup mechanisms:

```
2029 package org.oasisopen.sca.client;  
2030  
2031 public interface SCAClientFactoryFinder {  
2032  
2033     SCAClientFactory find(Properties properties,  
2034                          ClassLoader classLoader,  
2035                          URI domainURI )  
2036     throws NoSuchDomainException ;  
2037 }
```

2038 *Figure 9-11: SCAClientFactoryFinder interface*

2039 **find (Properties properties, ClassLoader classloader, URI domainURI) method:**

2040 Obtains an implementation of the SCAClientFactory interface.

2041 Returns:

- 2042 • **SCAClientFactory** implementation object
2043 The implementation of the SCAClientFactoryFinder.find method MUST return an object
2044 which is an implementation of the SCAClientFactory interface, for the SCA Domain
2045 represented by the doaminURI parameter, using the supplied properties and classloader.
2046 [JCA80055]

2047 Parameters:

- 2048 • **properties** - a set of Properties that can be used when creating the object which
2049 implements the SCAClientFactory interface.
- 2050 • **classLoader** - a ClassLoader to use when creating the object which implements the
2051 SCAClientFactory interface.
- 2052 • **domainURI** - a URI for the SCA Domain targeted by the SCAClientFactory

2053 Exceptions:

- 2054 • The implementation of the SCAClientFactoryFinder.find method MUST throw a
2055 ServiceRuntimeException if the SCAClientFactory implementation could not be found.
2056 [JCA80056]

2057 9.11 SCAClientFactoryFinderImpl Class

2058 This class is a default implementation of an SCAClientFactoryFinder, which is used to find an
2059 implementation of an SCAClientFactory subclass, as used to obtain an SCAClient object for use by
2060 a client. SCA runtime providers can replace this implementation with their own version.

```
2061 package org.oasisopen.sca.client.impl;  
2062  
2063 public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder  
2064 {  
2065     ...  
2066     public SCAClientFactoryFinderImpl() {...}  
2067  
2068     public SCAClientFactory find(Properties properties,  
2069                                ClassLoader classLoader  
2070                                URI domainURI)  
2071     throws NoSuchDomainException, ServiceRuntimeException {...}
```

```
2072     ...
2073 }
```

2074 *Figure 9-12: SCAClientFactoryFinderImpl class*

2075 **SCAClientFactoryFinderImpl () method:**

2076 Public constructor for the SCAClientFactoryFinderImpl.

2077 Returns:

- 2078 • **SCAClientFactoryFinderImpl** which implements the SCAClientFactoryFinder interface

2079 Parameters:

- 2080 • **none**

2081 Exceptions:

- 2082 • **none**

2083 **find (Properties, ClassLoader, URI) method:**

2084 Obtains an implementation of the SCAClientFactory interface. It discovers a provider's
2085 SCAClientFactory implementation by referring to the following information in this order:

- 2086 1. The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on
2087 the newInstance() method call if specified
- 2088 2. The org.oasisopen.sca.client.SCAClientFactory property from the System Properties
- 2089 3. The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file

2090 Returns:

- 2091 • **SCAClientFactory** implementation object

2092 Parameters:

- 2093 • **properties** - a set of Properties that can be used when creating the object which
2094 implements the SCAClientFactory interface.
- 2095 • **classLoader** - a ClassLoader to use when creating the object which implements the
2096 SCAClientFactory interface.
- 2097 • **domainURI** - a URI for the SCA Domain targeted by the SCAClientFactory

2098 Exceptions:

- 2099 • **ServiceRuntimeException** - if the SCAClientFactory implementation could not be found

2100 **9.12 NoSuchDomainException**

2101 Figure 9-13 shows the NoSuchDomainException:

```
2102 package org.oasisopen.sca;
2103
2104 public class NoSuchDomainException extends Exception {
2105     ...
2106 }
```

2107 *Figure 9-13: NoSuchDomainException class*

2108 This exception indicates that the Domain specified could not be found.

2109 **9.13 NoSuchServiceException**

2110 Figure 9-14 shows the NoSuchServiceException:

```
2111 package org.oasisopen.sca;
```

```
2112  
2113 public class NoSuchServiceException extends Exception {  
2114     ...  
2115 }
```

2116 *Figure 9-14: NoSuchServiceException class*

2117 This exception indicates that the service specified could not be found.

2118

2119

10 Java Annotations

2120

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

2121

2122

2123

2124

2125

2126

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 the sections "`@Property`" and "`@Reference`" constrain those definitions to constructor parameters. An SCA runtime MUST verify the proper use of all SCA annotations and if an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the invalid implementation code. [JCA90001]

2127

2128

2129

SCA annotations MUST NOT be used on static methods or on 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. [JCA90002]

2130

10.1 @AllowsPassByReference

2131

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

2132

2133

2134

2135

2136

2137

2138

2139

2140

2141

2142

2143

2144

2145

2146

2147

2148

```
package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({TYPE, METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface AllowsPassByReference {

    boolean value() default true;
}
```

2149

2150

2151

2152

The `@AllowsPassByReference` annotation allows service method implementations and client references to be marked as "allows pass by reference" to indicate that they use input parameters, return values and exceptions in a manner that allows the SCA runtime to avoid the cost of copying mutable objects when a remotable service is called locally within the same JVM.

2153

The `@AllowsPassByReference` annotation has the following attribute:

2154

2155

2156

- **value** – specifies whether the "allows pass by reference" marker applies to the service implementation class, service implementation method, or client reference to which this annotation applies; if not specified, defaults to true.

2157

The `@AllowsPassByReference` annotation MUST only annotate the following locations:

2158

a service implementation class

2159

an individual method of a remotable service implementation

2160

2161

- an individual reference which uses a remotable interface, where the reference is a field, a setter method, or a constructor parameter [JCA90052]

2162

2163

The "allows pass by reference" marking of a method implementation of a remotable service is determined as follows:

2164

2165

1. If the method has an `@AllowsPassByReference` annotation, the method is marked "allows pass by reference" if and only if the value of the method's annotation is true.

- 2166 2. Otherwise, if the class has an `@AllowsPassByReference` annotation, the method is marked
2167 "allows pass by reference" if and only if the value of the class's annotation is true.
2168 3. Otherwise, the method is not marked "allows pass by reference".

2169 The "allows pass by reference" marking of a reference for a remotable service is determined as
2170 follows:

- 2171 1. If the reference has an `@AllowsPassByReference` annotation, the reference is marked
2172 "allows pass by reference" if and only if the value of the reference's annotation is true.
2173 2. Otherwise, if the service implementation class containing the reference has an
2174 `@AllowsPassByReference` annotation, the reference is marked "allows pass by reference" if
2175 and only if the value of the class's annotation is true.
2176 3. Otherwise, the reference is not marked "allows pass by reference".

2177

2178 The following snippet shows a sample where `@AllowsPassByReference` is defined for the
2179 implementation of a service method on the Java component implementation class.

2180

```
2181 @AllowsPassByReference  
2182 public String hello(String message) {  
2183     ...  
2184 }  
2185
```

2186 The following snippet shows a sample where `@AllowsPassByReference` is defined for a client
2187 reference of a Java component implementation class.

```
2188 @AllowsPassByReference  
2189 @Reference  
2190 private StockQuoteService stockQuote;  
2191
```

2192 10.2 @AsyncFault

2193 The following Java code defines the **@AsyncFault** annotation:

2194

```
2195 package org.oasisopen.sca.annotation;  
2196  
2197 import static java.lang.annotation.ElementType.METHOD;  
2198 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2199 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2200  
2201 import java.lang.annotation.Inherited;  
2202 import java.lang.annotation.Retention;  
2203 import java.lang.annotation.Target;  
2204  
2205 @Inherited  
2206 @Target({METHOD})  
2207 @Retention(RUNTIME)  
2208 public @interface AsyncInvocation {  
2209  
2210     Class<?>[] value() default {};  
2211  
2212 }
```

2213 The **@AsyncInvocation** annotation is used to indicate the faults/exceptions which are returned by
2214 the asynchronous service method which it annotates.

2215

2216 10.3 @AsyncInvocation

2217 The following Java code defines the **@AsyncInvocation** annotation, which is used to attach the
2218 "asyncInvocation" policy intent to an interface or to a method:

```
2219 package org.oasisopen.sca.annotation;
2220
2221 import static java.lang.annotation.ElementType.METHOD;
2222 import static java.lang.annotation.ElementType.TYPE;
2223 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2224 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2225
2226 import java.lang.annotation.Inherited;
2227 import java.lang.annotation.Retention;
2228 import java.lang.annotation.Target;
2229
2230 @Inherited
2231 @Target({TYPE, METHOD})
2232 @Retention(RUNTIME)
2233 @Intent(AsyncInvocation.ASYNCINVOCATION)
2234 public @interface AsyncInvocation {
2235     String ASYNCINVOCATION = SCA_PREFIX + "asyncInvocation";
2236
2237     boolean value() default true;
2238 }
2239
```

2240 The **@AsyncInvocation** annotation is used to indicate that the operations of a Java interface uses
2241 the long-running request-response pattern as described in the SCA Assembly specification.

2242

2243 10.4 @Authentication

2244 The following Java code defines the **@Authentication** annotation:

```
2245 package org.oasisopen.sca.annotation;
2246
2247 import static java.lang.annotation.ElementType.FIELD;
2248 import static java.lang.annotation.ElementType.METHOD;
2249 import static java.lang.annotation.ElementType.PARAMETER;
2250 import static java.lang.annotation.ElementType.TYPE;
2251 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2252 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2253
2254 import java.lang.annotation.Inherited;
2255 import java.lang.annotation.Retention;
2256 import java.lang.annotation.Target;
2257
2258 @Inherited
2259 @Target({TYPE, FIELD, METHOD, PARAMETER})
2260 @Retention(RUNTIME)
2261 @Intent(Authentication.AUTHENTICATION)
2262 public @interface Authentication {
2263     String AUTHENTICATION = SCA_PREFIX + "authentication";
2264     String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";
2265     String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";
2266
2267     /**
2268
```

```

2269     * List of authentication qualifiers (such as "message"
2270     * or "transport").
2271     *
2272     * @return authentication qualifiers
2273     */
2274     @Qualifier
2275     String[] value() default "";
2276 }

```

2277 The **@Authentication** annotation is used to indicate the need for authentication. See the SCA
2278 Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section](#)
2279 [on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

2280 10.5 @Authorization

2281 The following Java code defines the @Authorization annotation:

```

2282 package org.oasisopen.sca.annotation;
2283
2284 import static java.lang.annotation.ElementType.FIELD;
2285 import static java.lang.annotation.ElementType.METHOD;
2286 import static java.lang.annotation.ElementType.PARAMETER;
2287 import static java.lang.annotation.ElementType.TYPE;
2288 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2289 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2290
2291 import java.lang.annotation.Inherited;
2292 import java.lang.annotation.Retention;
2293 import java.lang.annotation.Target;
2294
2295 /**
2296  * The @Authorization annotation is used to indicate that
2297  * an authorization policy is required.
2298  */
2299 @Inherited
2300 @Target({TYPE, FIELD, METHOD, PARAMETER})
2301 @Retention(RUNTIME)
2302 @Intent(Authorization.AUTHORIZATION)
2303 public @interface Authorization {
2304     String AUTHORIZATION = SCA_PREFIX + "authorization";
2305 }

```

2306 The **@Authorization** annotation is used to indicate the need for an authorization policy. See the
2307 SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the
2308 [section on Application of Intent Annotations](#) for samples of how intent annotations are used in
2309 Java.

2310 10.6 @Callback

2311 The following Java code defines the **@Callback** annotation:

```

2312
2313 package org.oasisopen.sca.annotation;
2314
2315 import static java.lang.annotation.ElementType.FIELD;
2316 import static java.lang.annotation.ElementType.METHOD;
2317 import static java.lang.annotation.ElementType.TYPE;
2318 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2319 import java.lang.annotation.Retention;

```

```

2320     import java.lang.annotation.Target;
2321
2322     @Target({TYPE, METHOD, FIELD})
2323     @Retention(RUNTIME)
2324     public @interface Callback {
2325
2326         Class<?> value() default Void.class;
2327     }
2328
2329

```

2330 The @Callback annotation is used to annotate a service interface or to annotate a Java class (used
2331 to define an interface) with a callback interface by specifying the Java class object of the callback
2332 interface as an attribute.

2333 The @Callback annotation has the following attribute:

- 2334 • **value** – the name of a Java class file containing the callback interface

2335

2336 The @Callback annotation can also be used to annotate a method or a field of an SCA
2337 implementation class, in order to have a callback object injected. When used to annotate a
2338 method or a field of an implementation class for injection of a callback object, the @Callback
2339 annotation MUST NOT specify any attributes. [JCA90046] When used to annotate a method or a
2340 field of an implementation class for injection of a callback object, the type of the method or field
2341 MUST be the callback interface of at least one bidirectional service offered by the implementation
2342 class. [JCA90054] When used to annotate a setter method or a field of an implementation class
2343 for injection of a callback object, the SCA runtime MUST inject a callback reference proxy into that
2344 method or field when the Java class is initialized, if the component is invoked via a service which
2345 has a callback interface and where the type of the setter method or field corresponds to the type
2346 of the callback interface. [JCA90058]

2347 The @Callback annotation MUST NOT appear on a setter method or a field of a Java
2348 implementation class that has COMPOSITE scope. [JCA90057]

2349 An example use of the @Callback annotation to declare a callback interface follows:

```

2350     package somepackage;
2351     import org.oasisopen.sca.annotation.Callback;
2352     import org.oasisopen.sca.annotation.Remotable;
2353     @Remotable
2354     @Callback(MyServiceCallback.class)
2355     public interface MyService {
2356
2357         void someMethod(String arg);
2358     }
2359
2360     @Remotable
2361     public interface MyServiceCallback {
2362
2363         void receiveResult(String result);
2364     }
2365

```

2366 In this example, the implied component type is:

```

2367     <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" >
2368
2369         <service name="MyService">
2370             <interface.java interface="somepackage.MyService"
2371                 callbackInterface="somepackage.MyServiceCallback"/>
2372         </service>
2373     </componentType>

```

2374 10.7 @ComponentName

2375 The following Java code defines the **@ComponentName** annotation:

2376

```
2377 package org.oasisopen.sca.annotation;
2378
2379 import static java.lang.annotation.ElementType.FIELD;
2380 import static java.lang.annotation.ElementType.METHOD;
2381 import static java.lang.annotation.ElementType.TYPE;
2382 import java.lang.annotation.Retention;
2383 import java.lang.annotation.Target;
2384
2385 @Target({METHOD, FIELD})
2386 @Retention(RUNTIME)
2387 public @interface ComponentName {
2388
2389 }
2390
```

2391 The @ComponentName annotation is used to denote a Java class field or setter method that is
2392 used to inject the component name.

2393 The following snippet shows a component name field definition sample.

2394

```
2395 @ComponentName
2396 private String componentName;
2397
```

2398 The following snippet shows a component name setter method sample.

2399

```
2400 @ComponentName
2401 public void setComponentName(String name) {
2402     //...
2403 }
```

2404 10.8 @Confidentiality

2405 The following Java code defines the **@Confidentiality** annotation:

2406

```
2407 package org.oasisopen.sca.annotation;
2408
2409 import static java.lang.annotation.ElementType.FIELD;
2410 import static java.lang.annotation.ElementType.METHOD;
2411 import static java.lang.annotation.ElementType.PARAMETER;
2412 import static java.lang.annotation.ElementType.TYPE;
2413 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2414 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2415
2416 import java.lang.annotation.Inherited;
2417 import java.lang.annotation.Retention;
2418 import java.lang.annotation.Target;
2419
2420 @Inherited
2421 @Target({TYPE, FIELD, METHOD, PARAMETER})
2422 @Retention(RUNTIME)
2423 @Intent(Confidentiality.CONFIDENTIALITY)
2424 public @interface Confidentiality {
```

```

2425     String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
2426     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
2427     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";
2428
2429     /**
2430      * List of confidentiality qualifiers such as "message" or
2431      * "transport".
2432      *
2433      * @return confidentiality qualifiers
2434      */
2435     @Qualifier
2436     String[] value() default "";
2437 }

```

2438 The **@Confidentiality** annotation is used to indicate the need for confidentiality. See the SCA Policy
2439 Framework Specification [POLICY] for details on the meaning of the intent. See the [section on](#)
2440 [Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

2441 10.9 @Constructor

2442 The following Java code defines the **@Constructor** annotation:

```

2443
2444     package org.oasisopen.sca.annotation;
2445
2446     import static java.lang.annotation.ElementType.CONSTRUCTOR;
2447     import static java.lang.annotation.RetentionPolicy.RUNTIME;
2448     import java.lang.annotation.Retention;
2449     import java.lang.annotation.Target;
2450
2451     @Target(CONSTRUCTOR)
2452     @Retention(RUNTIME)
2453     public @interface Constructor { }
2454

```

2455 The @Constructor annotation is used to mark a particular constructor to use when instantiating a
2456 Java component implementation. If a constructor of an implementation class is annotated with
2457 @Constructor and the constructor has parameters, each of these parameters MUST have either a
2458 @Property annotation or a @Reference annotation. [JCA90003]

2459 The following snippet shows a sample for the @Constructor annotation.

```

2460
2461     public class HelloServiceImpl implements HelloService {
2462
2463         public HelloServiceImpl(){
2464             ...
2465         }
2466
2467         @Constructor
2468         public HelloServiceImpl(@Property(name="someProperty")
2469                                 String someProperty ){
2470             ...
2471         }
2472
2473         public String hello(String message) {
2474             ...
2475         }
2476     }

```

2477 10.10 @Context

2478 The following Java code defines the **@Context** annotation:

2479

```
2480 package org.oasisopen.sca.annotation;
2481
2482 import static java.lang.annotation.ElementType.FIELD;
2483 import static java.lang.annotation.ElementType.METHOD;
2484 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2485 import java.lang.annotation.Retention;
2486 import java.lang.annotation.Target;
2487
2488 @Target({METHOD, FIELD})
2489 @Retention(RUNTIME)
2490 public @interface Context {
2491
2492 }
2493
```

2494 The @Context annotation is used to denote a Java class field or a setter method that is used to
2495 inject a composite context for the component. The type of context to be injected is defined by the
2496 type of the Java class field or type of the setter method input argument; the type is either
2497 **ComponentContext** or **RequestContext**.

2498 The @Context annotation has no attributes.

2499 The following snippet shows a ComponentContext field definition sample.

2500

```
2501 @Context
2502 protected ComponentContext context;
2503
```

2504 The following snippet shows a RequestContext field definition sample.

2505

```
2506 @Context
2507 protected RequestContext context;
```

2508 10.11 @Destroy

2509 The following Java code defines the **@Destroy** annotation:

2510

```
2511 package org.oasisopen.sca.annotation;
2512
2513 import static java.lang.annotation.ElementType.METHOD;
2514 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2515 import java.lang.annotation.Retention;
2516 import java.lang.annotation.Target;
2517
2518 @Target(METHOD)
2519 @Retention(RUNTIME)
2520 public @interface Destroy {
2521
2522 }
2523
```

2524 The @Destroy annotation is used to denote a single Java class method that will be called when the
2525 scope defined for the implementation class ends. A method annotated with @Destroy can have
2526 any access modifier and MUST have a void return type and no arguments. [JCA90004]

2527 If there is a method annotated with @Destroy that matches the criteria for the annotation, the
2528 SCA runtime MUST call the annotated method when the scope defined for the implementation
2529 class ends. [JCA90005]

2530 The following snippet shows a sample for a destroy method definition.

```
2531  
2532 @Destroy  
2533 public void myDestroyMethod() {  
2534     ...  
2535 }
```

2536 10.12 @EagerInit

2537 The following Java code defines the @EagerInit annotation:

```
2538  
2539 package org.oasisopen.sca.annotation;  
2540  
2541 import static java.lang.annotation.ElementType.TYPE;  
2542 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2543 import java.lang.annotation.Retention;  
2544 import java.lang.annotation.Target;  
2545  
2546 @Target(TYPE)  
2547 @Retention(RUNTIME)  
2548 public @interface EagerInit {  
2549  
2550 }  
2551
```

2552 The @EagerInit annotation is used to mark the Java class of a COMPOSITE scoped
2553 implementation for eager initialization. When marked for eager initialization with an @EagerInit
2554 annotation, the composite scoped instance MUST be created when its containing component is
2555 started. [JCA90007]

2556 10.13 @Init

2557 The following Java code defines the @Init annotation:

```
2558  
2559 package org.oasisopen.sca.annotation;  
2560  
2561 import static java.lang.annotation.ElementType.METHOD;  
2562 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2563 import java.lang.annotation.Retention;  
2564 import java.lang.annotation.Target;  
2565  
2566 @Target(METHOD)  
2567 @Retention(RUNTIME)  
2568 public @interface Init {  
2569  
2570 }  
2571  
2572
```

2573 The @Init annotation is used to denote a single Java class method that is called when the scope
2574 defined for the implementation class starts. A method marked with the @Init annotation can have
2575 any access modifier and MUST have a void return type and no arguments. [JCA90008]

2576 If there is a method annotated with @Init that matches the criteria for the annotation, the SCA
2577 runtime MUST call the annotated method after all property and reference injection is complete.
2578 [JCA90009]

2579 The following snippet shows an example of an init method definition.

```
2580  
2581 @Init  
2582 public void myInitMethod() {  
2583     ...  
2584 }
```

2585 10.14 @Integrity

2586 The following Java code defines the **@Integrity** annotation:

```
2587 package org.oasisopen.sca.annotation;  
2588  
2589 import static java.lang.annotation.ElementType.FIELD;  
2590 import static java.lang.annotation.ElementType.METHOD;  
2591 import static java.lang.annotation.ElementType.PARAMETER;  
2592 import static java.lang.annotation.ElementType.TYPE;  
2593 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2594 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2595  
2596 import java.lang.annotation.Inherited;  
2597 import java.lang.annotation.Retention;  
2598 import java.lang.annotation.Target;  
2600  
2601 @Inherited  
2602 @Target({TYPE, FIELD, METHOD, PARAMETER})  
2603 @Retention(RUNTIME)  
2604 @Intent(Integrity.INTEGRITY)  
2605 public @interface Integrity {  
2606     String INTEGRITY = SCA_PREFIX + "integrity";  
2607     String INTEGRITY_MESSAGE = INTEGRITY + ".message";  
2608     String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";  
2609  
2610     /**  
2611      * List of integrity qualifiers (such as "message" or "transport").  
2612      *  
2613      * @return integrity qualifiers  
2614      */  
2615     @Qualifier  
2616     String[] value() default "";  
2617 }  
2618
```

2619 The **@Integrity** annotation is used to indicate that the invocation requires integrity (i.e. no
2620 tampering of the messages between client and service). See the SCA Policy Framework
2621 Specification [POLICY] for details on the meaning of the intent. See the [section on Application of
2622 Intent Annotations](#) for samples of how intent annotations are used in Java.

2623 10.15 @Intent

2624 The following Java code defines the **@Intent** annotation:

```

2625
2626 package org.oasisopen.sca.annotation;
2627
2628 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
2629 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2630 import java.lang.annotation.Retention;
2631 import java.lang.annotation.Target;
2632
2633 @Target({ANNOTATION_TYPE})
2634 @Retention(RUNTIME)
2635 public @interface Intent {
2636     /**
2637      * The qualified name of the intent, in the form defined by
2638      * {@link javax.xml.namespace.QName#toString}.
2639      * @return the qualified name of the intent
2640      */
2641     String value() default "";
2642
2643     /**
2644      * The XML namespace for the intent.
2645      * @return the XML namespace for the intent
2646      */
2647     String targetNamespace() default "";
2648
2649     /**
2650      * The name of the intent within its namespace.
2651      * @return name of the intent within its namespace
2652      */
2653     String localPart() default "";
2654 }
2655

```

2656 The @Intent annotation is used for the creation of new annotations for specific intents. It is not
2657 expected that the @Intent annotation will be used in application code.

2658 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to
2659 define new intent annotations.

2660 10.16 @ManagedSharedTransaction

2661 The following Java code defines the @ManagedSharedTransaction annotation:

```

2662 package org.oasisopen.sca.annotation;
2663
2664 import static java.lang.annotation.ElementType.FIELD;
2665 import static java.lang.annotation.ElementType.METHOD;
2666 import static java.lang.annotation.ElementType.PARAMETER;
2667 import static java.lang.annotation.ElementType.TYPE;
2668 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2669 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2670
2671 import java.lang.annotation.Inherited;
2672 import java.lang.annotation.Retention;
2673 import java.lang.annotation.Target;
2674
2675 /**
2676  * The @ManagedSharedTransaction annotation is used to indicate that
2677  * a distributed ACID transaction is required.
2678  */
2679 @Inherited

```

```

2680 @Target({TYPE, FIELD, METHOD, PARAMETER})
2681 @Retention(RUNTIME)
2682 @Intent(ManagedSharedTransaction.MANAGEDSHAREDTRANSACTION)
2683 public @interface ManagedSharedTransaction {
2684     String MANAGEDSHAREDTRANSACTION = SCA_PREFIX +
2685     "managedSharedTransaction";
2686 }

```

2687 The **@ManagedSharedTransaction** annotation is used to indicate the need for a distributed and
2688 globally coordinated ACID transaction. See the SCA Policy Framework Specification [POLICY] for
2689 details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for
2690 samples of how intent annotations are used in Java.

2691 10.17 @ManagedTransaction

2692 The following Java code defines the @ManagedTransaction annotation:

```

2693 import static java.lang.annotation.ElementType.FIELD;
2694 import static java.lang.annotation.ElementType.METHOD;
2695 import static java.lang.annotation.ElementType.PARAMETER;
2696 import static java.lang.annotation.ElementType.TYPE;
2697 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2698 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2699
2700 import java.lang.annotation.Inherited;
2701 import java.lang.annotation.Retention;
2702 import java.lang.annotation.Target;
2703
2704 /**
2705  * The @ManagedTransaction annotation is used to indicate the
2706  * need for an ACID transaction environment.
2707  */
2708 @Inherited
2709 @Target({TYPE, FIELD, METHOD, PARAMETER})
2710 @Retention(RUNTIME)
2711 @Intent(ManagedTransaction.MANAGEDTRANSACTION)
2712 public @interface ManagedTransaction {
2713     String MANAGEDTRANSACTION = SCA_PREFIX + "managedTransaction";
2714     String MANAGEDTRANSACTION_MESSAGE = MANAGEDTRANSACTION + ".local";
2715     String MANAGEDTRANSACTION_TRANSPORT = MANAGEDTRANSACTION + ".global";
2716
2717     /**
2718      * List of managedTransaction qualifiers (such as "global" or
2719      * "local").
2720      *
2721      * @return managedTransaction qualifiers
2722      */
2723     @Qualifier
2724     String[] value() default "";
2725 }

```

2726 The **@ManagedTransaction** annotation is used to indicate the need for an ACID transaction. See
2727 the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the
2728 [section on Application of Intent Annotations](#) for samples of how intent annotations are used in
2729 Java.

2730 10.18 @MutualAuthentication

2731 The following Java code defines the @MutualAuthentication annotation:

```

2732 package org.oasisopen.sca.annotation;
2733
2734 import static java.lang.annotation.ElementType.FIELD;
2735 import static java.lang.annotation.ElementType.METHOD;
2736 import static java.lang.annotation.ElementType.PARAMETER;
2737 import static java.lang.annotation.ElementType.TYPE;
2738 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2739 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2740
2741 import java.lang.annotation.Inherited;
2742 import java.lang.annotation.Retention;
2743 import java.lang.annotation.Target;
2744
2745 /**
2746  * The @MutualAuthentication annotation is used to indicate that
2747  * a mutual authentication policy is needed.
2748  */
2749 @Inherited
2750 @Target({TYPE, FIELD, METHOD, PARAMETER})
2751 @Retention(RUNTIME)
2752 @Intent(MutualAuthentication.MUTUALAUTHENTICATION)
2753 public @interface MutualAuthentication {
2754     String MUTUALAUTHENTICATION = SCA_PREFIX + "mutualAuthentication";
2755 }

```

2756 The **@MutualAuthentication** annotation is used to indicate the need for mutual authentication
2757 between a service consumer and a service provider. See the SCA Policy Framework Specification
2758 [POLICY] for details on the meaning of the intent. See the [section on Application of Intent](#)
2759 [Annotations](#) for samples of how intent annotations are used in Java.

2760 10.19 @NoManagedTransaction

2761 The following Java code defines the @NoManagedTransaction annotation:

```

2762 package org.oasisopen.sca.annotation;
2763
2764 import static java.lang.annotation.ElementType.FIELD;
2765 import static java.lang.annotation.ElementType.METHOD;
2766 import static java.lang.annotation.ElementType.PARAMETER;
2767 import static java.lang.annotation.ElementType.TYPE;
2768 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2769 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2770
2771 import java.lang.annotation.Inherited;
2772 import java.lang.annotation.Retention;
2773 import java.lang.annotation.Target;
2774
2775 /**
2776  * The @NoManagedTransaction annotation is used to indicate that
2777  * a non-transactional environment is needed.
2778  */
2779 @Inherited
2780 @Target({TYPE, FIELD, METHOD, PARAMETER})
2781 @Retention(RUNTIME)
2782 @Intent(NoManagedTransaction.NOMANAGEDTRANSACTION)
2783 public @interface NoManagedTransaction {
2784     String NOMANAGEDTRANSACTION = SCA_PREFIX + "noManagedTransaction";
2785 }

```

2786 The **@NoManagedTransaction** annotation is used to indicate that the component does not want
2787 to run in an ACID transaction. See the SCA Policy Framework Specification [POLICY] for details on
2788 the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how
2789 intent annotations are used in Java.

2790 10.20 @OneWay

2791 The following Java code defines the **@OneWay** annotation:

2792

```
2793 package org.oasisopen.sca.annotation;  
2794  
2795 import static java.lang.annotation.ElementType.METHOD;  
2796 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2797 import java.lang.annotation.Retention;  
2798 import java.lang.annotation.Target;  
2799  
2800 @Target(METHOD)  
2801 @Retention(RUNTIME)  
2802 public @interface OneWay {  
2803  
2804  
2805 }  
2806
```

2807 A method annotated with **@OneWay** MUST have a void return type and MUST NOT have declared
2808 checked exceptions. [JCA90055]

2809 When a method of a Java interface is annotated with **@OneWay**, the SCA runtime MUST ensure
2810 that all invocations of that method are executed in a non-blocking fashion, as described in the
2811 [section on Asynchronous Programming](#). [JCA90056]

2812 The **@OneWay** annotation has no attributes.

2813 The following snippet shows the use of the **@OneWay** annotation on an interface.

```
2814 package services.hello;  
2815  
2816 import org.oasisopen.sca.annotation.OneWay;  
2817  
2818 public interface HelloService {  
2819     @OneWay  
2820     void hello(String name);  
2821 }
```

2822 10.21 @PolicySets

2823 The following Java code defines the **@PolicySets** annotation:

2824

```
2825 package org.oasisopen.sca.annotation;  
2826  
2827 import static java.lang.annotation.ElementType.FIELD;  
2828 import static java.lang.annotation.ElementType.METHOD;  
2829 import static java.lang.annotation.ElementType.PARAMETER;  
2830 import static java.lang.annotation.ElementType.TYPE;  
2831 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2832  
2833 import java.lang.annotation.Retention;  
2834 import java.lang.annotation.Target;  
2835
```

```

2836 @Target({TYPE, FIELD, METHOD, PARAMETER})
2837 @Retention(RUNTIME)
2838 public @interface PolicySets {
2839     /**
2840      * Returns the policy sets to be applied.
2841      *
2842      * @return the policy sets to be applied
2843      */
2844     String[] value() default "";
2845 }
2846

```

2847 The **@PolicySets** annotation is used to attach one or more SCA Policy Sets to a Java
2848 implementation class or to one of its subelements.

2849 See the [section "Policy Set Annotations"](#) for details and samples.

2850 10.22 @Property

2851 The following Java code defines the **@Property** annotation:

```

2852 package org.oasisopen.sca.annotation;
2853
2854 import static java.lang.annotation.ElementType.FIELD;
2855 import static java.lang.annotation.ElementType.METHOD;
2856 import static java.lang.annotation.ElementType.PARAMETER;
2857 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2858 import java.lang.annotation.Retention;
2859 import java.lang.annotation.Target;
2860
2861 @Target({METHOD, FIELD, PARAMETER})
2862 @Retention(RUNTIME)
2863 public @interface Property {
2864
2865     String name() default "";
2866     boolean required() default true;
2867 }
2868

```

2869 The @Property annotation is used to denote a Java class field, a setter method, or a constructor
2870 parameter that is used to inject an SCA property value. The type of the property injected, which
2871 can be a simple Java type or a complex Java type, is defined by the type of the Java class field or
2872 the type of the input parameter of the setter method or constructor.

2873 When the Java type of a field, setter method or constructor parameter with the @Property
2874 annotation is a primitive type or a JAXB annotated class, the SCA runtime MUST convert a
2875 property value specified by an SCA component definition into an instance of the Java type as
2876 defined by the XML to Java mapping in the JAXB specification [JAXB] with XML schema validation
2877 enabled. [JCA90061]

2878 When the Java type of a field, setter method or constructor parameter with the @Property
2879 annotation is not a JAXB annotated class, the SCA runtime can use any XML to Java mapping
2880 when converting property values into instances of the Java type.

2881 The @Property annotation MUST NOT be used on a class field that is declared as final. [JCA90011]

2882 Where there is both a setter method and a field for a property, the setter method is used.

2883 The @Property annotation has the following attributes:

- 2884 • **name (optional)** – the name of the property. For a field annotation, the default is the
2885 name of the field of the Java class. For a setter method annotation, the default is the
2886 JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a

2887 @Property annotation applied to a constructor parameter, there is no default value for the
2888 name attribute and the name attribute MUST be present. [JCA90013]

- 2889 • **required (optional)** – a boolean value which specifies whether injection of the property
2890 value is required or not, where true means injection is required and false means injection
2891 is not required. Defaults to true. For a @Property annotation applied to a constructor
2892 parameter, the required attribute MUST NOT have the value false. [JCA90014]

2893

2894 The following snippet shows a property field definition sample.

2895

```
2896 @Property(name="currency", required=true)  
2897 protected String currency;
```

2898

2899 The following snippet shows a property setter sample

2900

```
2901 @Property(name="currency", required=true)  
2902 public void setCurrency( String theCurrency ) {  
2903     ....  
2904 }
```

2905

2906 For a @Property annotation, if the type of the Java class field or the type of the input parameter of
2907 the setter method or constructor is defined as an array or as any type that extends or implements
2908 java.util.Collection, then the SCA runtime MUST introspect the component type of the
2909 implementation with a <property/> element with a @many attribute set to true, otherwise
2910 @many MUST be set to false. [JCA90047]

2911 The following snippet shows the definition of a configuration property using the @Property
2912 annotation for a collection.

```
2913 ...  
2914 private List<String> helloConfigurationProperty;  
2915  
2916 @Property(required=true)  
2917 public void setHelloConfigurationProperty(List<String> property) {  
2918     helloConfigurationProperty = property;  
2919 }  
2920 ...
```

2921 10.23 @Qualifier

2922 The following Java code defines the @Qualifier annotation:

2923

```
2924 package org.oasisopen.sca.annotation;  
2925  
2926 import static java.lang.annotation.ElementType.METHOD;  
2927 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2928  
2929 import java.lang.annotation.Retention;  
2930 import java.lang.annotation.Target;  
2931  
2932 @Target(METHOD)  
2933 @Retention(RUNTIME)  
2934 public @interface Qualifier {
```

2935 }

2936

2937 The @Qualifier annotation is applied to an attribute of a specific intent annotation definition,
2938 defined using the @Intent annotation, to indicate that the attribute provides qualifiers for the
2939 intent. The @Qualifier annotation MUST be used in a specific intent annotation definition where the
2940 intent has qualifiers. [JCA90015]

2941 See the section "How to Create Specific Intent Annotations" for details and samples of how to
2942 define new intent annotations.

2943 10.24 @Reference

2944 The following Java code defines the @Reference annotation:

2945

```
2946 package org.oasisopen.sca.annotation;  
2947  
2948 import static java.lang.annotation.ElementType.FIELD;  
2949 import static java.lang.annotation.ElementType.METHOD;  
2950 import static java.lang.annotation.ElementType.PARAMETER;  
2951 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2952 import java.lang.annotation.Retention;  
2953 import java.lang.annotation.Target;  
2954 @Target({METHOD, FIELD, PARAMETER})  
2955 @Retention(RUNTIME)  
2956 public @interface Reference {  
2957  
2958     String name() default "";  
2959     boolean required() default true;  
2960 }  
2961
```

2962 The @Reference annotation type is used to annotate a Java class field, a setter method, or a
2963 constructor parameter that is used to inject a service that resolves the reference. The interface of
2964 the service injected is defined by the type of the Java class field or the type of the input parameter
2965 of the setter method or constructor.

2966 The @Reference annotation MUST NOT be used on a class field that is declared as final.
2967 [JCA90016]

2968 Where there is both a setter method and a field for a reference, the setter method is used.

2969 The @Reference annotation has the following attributes:

- 2970 • **name : String (optional)** – the name of the reference. For a field annotation, the default is
2971 the name of the field of the Java class. For a setter method annotation, the default is the
2972 JavaBeans property name corresponding to the setter method name. For a @Reference
2973 annotation applied to a constructor parameter, there is no default for the name attribute
2974 and the name attribute MUST be present. [JCA90018]
- 2975 • **required (optional)** – a boolean value which specifies whether injection of the service
2976 reference is required or not, where true means injection is required and false means
2977 injection is not required. Defaults to true. For a @Reference annotation applied to a
2978 constructor parameter, the required attribute MUST have the value true. [JCA90019]

2979

2980 The following snippet shows a reference field definition sample.

2981

```
2982 @Reference(name="stockQuote", required=true)  
2983 protected StockQuoteService stockQuote;
```

2984

2985 The following snippet shows a reference setter sample

2986

```
2987 @Reference(name="stockQuote", required=true)
2988 public void setStockQuote( StockQuoteService theSQService ) {
2989     ...
2990 }
```

2991

2992 The following fragment from a component implementation shows a sample of a service reference
2993 using the @Reference annotation. The name of the reference is "helloService" and its type is
2994 HelloService. The clientMethod() calls the "hello" operation of the service referenced by the
2995 helloService reference.

2996

```
2997 package services.hello;
2998
2999 private HelloService helloService;
3000
3001 @Reference(name="helloService", required=true)
3002 public setHelloService(HelloService service) {
3003     helloService = service;
3004 }
3005
3006 public void clientMethod() {
3007     String result = helloService.hello("Hello World!");
3008     ...
3009 }
3010
```

3011 The presence of a @Reference annotation is reflected in the componentType information that the
3012 runtime generates through reflection on the implementation class. The following snippet shows
3013 the component type for the above component implementation fragment.

3014

```
3015 <?xml version="1.0" encoding="ASCII"?>
3016 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">
3017     <!-- Any services offered by the component would be listed here -->
3018     <reference name="helloService" multiplicity="1..1">
3019         <interface.java interface="services.hello.HelloService"/>
3020     </reference>
3021 </componentType>
3022
3023
3024
```

3025 If the type of a reference is not an array or any type that extends or implements
3026 java.util.Collection, then the SCA runtime MUST introspect the component type of the
3027 implementation with a <reference/> element with @multiplicity= 0..1 if the @Reference
3028 annotation required attribute is false and with @multiplicity=1..1 if the @Reference annotation
3029 required attribute is true. [JCA90020]

3030 If the type of a reference is defined as an array or as any type that extends or implements
3031 java.util.Collection, then the SCA runtime MUST introspect the component type of the
3032 implementation with a <reference/> element with @multiplicity=0..n if the @Reference
3033 annotation required attribute is false and with @multiplicity=1..n if the @Reference annotation
3034 required attribute is true. [JCA90021]

3035 The following fragment from a component implementation shows a sample of a service reference
3036 definition using the @Reference annotation on a java.util.List. The name of the reference is
3037 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
3038 services referenced by the helloServices reference. In this case, at least one HelloService needs
3039 to be present, so **required** is true.

```
3040
3041     @Reference(name="helloServices", required=true)
3042     protected List<HelloService> helloServices;
3043
3044     public void clientMethod() {
3045
3046         ...
3047         for (int index = 0; index < helloServices.size(); index++) {
3048             HelloService helloService =
3049                 (HelloService)helloServices.get(index);
3050             String result = helloService.hello("Hello World!");
3051         }
3052         ...
3053     }
3054
```

3055 The following snippet shows the XML representation of the component type reflected from for the
3056 former component implementation fragment. There is no need to author this component type in
3057 this case since it can be reflected from the Java class.

```
3058
3059 <?xml version="1.0" encoding="ASCII"?>
3060 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">
3061
3062     <!-- Any services offered by the component would be listed here -->
3063     <reference name="helloServices" multiplicity="1..n">
3064         <interface.java interface="services.hello.HelloService"/>
3065     </reference>
3066
3067 </componentType>
3068
```

3069 An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by
3070 the SCA runtime as null [JCA90022] An unwired reference with a multiplicity of 0..n MUST be
3071 presented to the implementation code by the SCA runtime as an empty array or empty collection
3072 [JCA90023]

3073 10.24.1 Reinjection

3074 References MAY be reinjected by an SCA runtime after the initial creation of a component if the
3075 reference target changes due to a change in wiring that has occurred since the component was
3076 initialized. [JCA90024]

3077 In order for reinjection to occur, the following MUST be true:

- 3078 1. The component MUST NOT be STATELESS scoped.
- 3079 2. The reference MUST use either field-based injection or setter injection. References that
3080 are injected through constructor injection MUST NOT be changed.

3081 [JCA90025]

3082 Setter injection allows for code in the setter method to perform processing in reaction to a change.

3083 If a reference target changes and the reference is not reinjected, the reference MUST continue to
3084 work as if the reference target was not changed. [JCA90026]

3085 If an operation is called on a reference where the target of that reference has been undeployed,
 3086 the SCA runtime SHOULD throw an InvalidServiceException. [JCA90027] If an operation is called
 3087 on a reference where the target of the reference has become unavailable for some reason, the
 3088 SCA runtime SHOULD throw a ServiceUnavailableException. [JCA90028] If the target service of
 3089 the reference is changed, the reference MUST either continue to work or throw an
 3090 InvalidServiceException when it is invoked. [JCA90029] If it doesn't work, the exception thrown
 3091 will depend on the runtime and the cause of the failure.

3092 A ServiceReference that has been obtained from a reference by ComponentContext.cast()
 3093 corresponds to the reference that is passed as a parameter to cast(). If the reference is
 3094 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue
 3095 to work as if the reference target was not changed. [JCA90030] If the target of a ServiceReference
 3096 has been undeployed, the SCA runtime SHOULD throw a InvalidServiceException when an
 3097 operation is invoked on the ServiceReference. [JCA90031] If the target of a ServiceReference has
 3098 become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an
 3099 operation is invoked on the ServiceReference. [JCA90032] If the target service of a
 3100 ServiceReference is changed, the reference MUST either continue to work or throw an
 3101 InvalidServiceException when it is invoked. [JCA90033] If it doesn't work, the exception thrown
 3102 will depend on the runtime and the cause of the failure.

3103 A reference or ServiceReference accessed through the component context by calling getService()
 3104 or getServiceReference() MUST correspond to the current configuration of the domain. This applies
 3105 whether or not reinjection has taken place. [JCA90034] If the target of a reference or
 3106 ServiceReference accessed through the component context by calling getService() or
 3107 getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a
 3108 reference to the undeployed or unavailable service, and attempts to call business methods
 3109 SHOULD throw an InvalidServiceException or a ServiceUnavailableException. [JCA90035] If the
 3110 target service of a reference or ServiceReference accessed through the component context by
 3111 calling getService() or getServiceReference() has changed, the returned value SHOULD be a
 3112 reference to the changed service. [JCA90036]

3113 The rules for reference reinjection also apply to references with a multiplicity of 0..n or 1..n. This
 3114 means that in the cases where reference reinjection is not allowed, the array or Collection for a
 3115 reference of multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes
 3116 occur to the reference wiring or to the targets of the wiring. [JCA90037] In cases where the
 3117 contents of a reference array or collection change when the wiring changes or the targets change,
 3118 then for references that use setter injection, the setter method MUST be called by the SCA
 3119 runtime for any change to the contents. [JCA90038] A reinjected array or Collection for a
 3120 reference MUST NOT be the same array or Collection object previously injected to the component.
 3121 [JCA90039]

3122

Change event	Effect on		
	Injected Reference or ServiceReference	Existing ServiceReference Object**	Subsequent invocations of ComponentContext.getServiceReference() or getService()
Change to the target of the reference	can be reinjected (if other conditions* apply). If not reinjected, then it continues to work as if the reference target was not changed.	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 throw InvalidServiceException.	Business methods throw InvalidServiceException.	Result is a reference to the undeployed service. Business methods throw InvalidServiceException.
Target service	Business methods throw ServiceUnavailableExce	Business methods throw ServiceUnavailableExce	Result is be a reference to the unavailable service. Business

becomes unavailable	ption	ption	methods throw ServiceUnavailableException.
Target service changed	might 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.	might 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 is a reference to the changed service.
<p>* Other conditions:</p> <p>The component cannot be STATELESS scoped.</p> <p>The reference has to use either field-based injection or setter injection. References that are injected through constructor injection cannot be changed.</p> <p>** Result of invoking ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().</p>			

3123

3124 10.25 @Remotable

3125 The following Java code defines the **@Remotable** annotation:

3126

3127 `package org.oasisopen.sca.annotation;`

3128

3129 `import static java.lang.annotation.ElementType.TYPE;`

3130 `import static java.lang.annotation.RetentionPolicy.RUNTIME;`

3131 `import java.lang.annotation.Retention;`

3132 `import java.lang.annotation.Target;`

3133

3134

3135 `@Target (TYPE)`

3136 `@Retention (RUNTIME)`

3137 `public @interface Remotable {`

3138

3139

3140

3141 The @Remotable annotation is used to indicate that an SCA service interface is remotable. The
3142 @Remotable annotation is valid only on a Java interface, a Java class, a field, a setter method, or
3143 a constructor parameter. It MUST NOT appear anywhere else. [JCA90053] A remotable service
3144 can be published externally as a service and MUST be translatable into a WSDL portType.
3145 [JCA90040]

3146 The @Remotable annotation has no attributes. When placed on a Java service interface, it
3147 indicates that the interface is remotable. When placed on a Java service implementation class, it
3148 indicates that all SCA service interfaces provided by the class (including the class itself, if the class
3149 defines an SCA service interface) are remotable. When placed on a service reference, it indicates
3150 that the interface for the reference is remotable.

3151 The following snippet shows the Java interface for a remotable service with its @Remotable
3152 annotation.

3153 `package services.hello;`

3154

3155 `import org.oasisopen.sca.annotation.*;`

```
3156
3157 @Remotable
3158 public interface HelloService {
3159
3160     String hello(String message);
3161 }
3162
```

3163 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
3164 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

3165 Complex data types exchanged via remotable service interfaces need to be compatible with the
3166 marshalling technology used by the service binding. For example, if the service is going to be
3167 exposed using the standard Web Service binding, then the parameters can be JAXB [JAX-B] types
3168 or they can be Service Data Objects (SDOs) [SDO].

3169 Independent of whether the remotable service is called from outside of the composite that
3170 contains it or from another component in the same composite, the data exchange semantics are
3171 **by-value**.

3172 Implementations of remotable services can modify input data during or after an invocation and
3173 can modify return data after the invocation. If a remotable service is called locally or remotely, the
3174 SCA container is responsible for making sure that no modification of input data or post-invocation
3175 modifications to return data are seen by the caller.

3176 The following snippet shows how a Java service implementation class can use the @Remotable
3177 annotation to define a remotable SCA service interface using a Java service interface that is not
3178 marked as remotable.

```
3179
3180 package services.hello;
3181
3182 import org.oasisopen.sca.annotation.*;
3183
3184 public interface HelloService {
3185
3186     String hello(String message);
3187 }
3188
3189 package services.hello;
3190
3191 import org.oasisopen.sca.annotation.*;
3192
3193 @Remotable
3194 @Service(HelloService.class)
3195 public class HelloServiceImpl implements HelloService {
3196
3197     public String hello(String message) {
3198         ...
3199     }
3200 }
3201
```

3202 The following snippet shows how a reference can use the @Remotable annotation to define a
3203 remotable SCA service interface using a Java service interface that is not marked as remotable.

```
3204
3205 package services.hello;
3206
3207 import org.oasisopen.sca.annotation.*;
3208
3209 public interface HelloService {
```

```

3210     String hello(String message);
3211 }
3212
3213
3214 package services.hello;
3215
3216 import org.oasisopen.sca.annotation.*;
3217
3218 public class HelloClient {
3219
3220     @Remotable
3221     @Reference
3222     protected HelloService myHello;
3223
3224     public String greeting(String message) {
3225         return myHello.hello(message);
3226     }
3227 }
3228

```

3229 10.26 @Requires

3230 The following Java code defines the **@Requires** annotation:

```

3231 package org.oasisopen.sca.annotation;
3232
3233 import static java.lang.annotation.ElementType.FIELD;
3234 import static java.lang.annotation.ElementType.METHOD;
3235 import static java.lang.annotation.ElementType.PARAMETER;
3236 import static java.lang.annotation.ElementType.TYPE;
3237 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3238
3239 import java.lang.annotation.Inherited;
3240 import java.lang.annotation.Retention;
3241 import java.lang.annotation.Target;
3242
3243 @Inherited
3244 @Retention(RUNTIME)
3245 @Target({TYPE, METHOD, FIELD, PARAMETER})
3246 public @interface Requires {
3247     /**
3248      * Returns the attached intents.
3249      *
3250      * @return the attached intents
3251      */
3252     String[] value() default "";
3253 }
3254

```

3256 The **@Requires** annotation supports general purpose intents specified as strings. Users can also
3257 define specific intent annotations using the @Intent annotation.

3258 See the [section "General Intent Annotations"](#) for details and samples.

3259 10.27 @Scope

3260 The following Java code defines the **@Scope** annotation:

```

3261 package org.oasisopen.sca.annotation;

```

```

3262
3263 import static java.lang.annotation.ElementType.TYPE;
3264 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3265 import java.lang.annotation.Retention;
3266 import java.lang.annotation.Target;
3267
3268 @Target(TYPE)
3269 @Retention(RUNTIME)
3270 public @interface Scope {
3271
3272     String value() default "STATELESS";
3273 }
3274 The @Scope annotation MUST only be used on a service's implementation class. It is an error to
3275 use this annotation on an interface. [JCA90041]

```

3276 The @Scope annotation has the following attribute:

- 3277 • **value** – the name of the scope.
- 3278 SCA defines the following scope names, but others can be defined by particular Java-
- 3279 based implementation types:
- 3280 STATELESS
- 3281 COMPOSITE
- 3282

3283 The default value is STATELESS.

3284 The following snippet shows a sample for a COMPOSITE scoped service implementation:

```

3285 package services.hello;
3286
3287 import org.oasisopen.sca.annotation.*;
3288
3289 @Service(HelloService.class)
3290 @Scope("COMPOSITE")
3291 public class HelloServiceImpl implements HelloService {
3292
3293     public String hello(String message) {
3294         ...
3295     }
3296 }
3297

```

3298 10.28 @Service

3299 The following Java code defines the **@Service** annotation:

```

3300 package org.oasisopen.sca.annotation;
3301
3302 import static java.lang.annotation.ElementType.TYPE;
3303 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3304 import java.lang.annotation.Retention;
3305 import java.lang.annotation.Target;
3306
3307 @Target(TYPE)
3308 @Retention(RUNTIME)
3309 public @interface Service {
3310
3311     Class<?>[] value();
3312     String[] names() default {};
3313 }
3314

```

3315 The @Service annotation is used on a component implementation class to specify the SCA services
3316 offered by the implementation. An implementation class need not be declared as implementing all
3317 of the interfaces implied by the services declared in its @Service annotation, but all methods of all
3318 the declared service interfaces MUST be present. [JCA90042] A class used as the implementation
3319 of a service is not required to have a @Service annotation. If a class has no @Service annotation,
3320 then the rules determining which services are offered and what interfaces those services have are
3321 determined by the specific implementation type.

3322 The @Service annotation has the following attributes:

- 3323 • **value (1..1)** – An array of interface or class objects that are exposed as services by this
3324 implementation. If the array is empty, no services are exposed.
- 3325 • **names (0..1)** - An array of Strings which are used as the service names for each of the
3326 interfaces declared in the **value** array. The number of Strings in the names attribute array
3327 of the @Service annotation MUST match the number of elements in the value attribute
3328 array. [JCA90050] The value of each element in the @Service names array MUST be
3329 unique amongst all the other element values in the array. [JCA90060]

3330 The **service name** of an exposed service defaults to the name of its interface or class, without the
3331 package name. If the names attribute is specified, the service name for each interface or class in
3332 the value attribute array is the String declared in the corresponding position in the names
3333 attribute array.

3334 If a component implementation has two services with the same Java simple name, the names
3335 attribute of the @Service annotation MUST be specified. [JCA90045] If a Java implementation
3336 needs to realize two services with the same Java simple name then this can be achieved through
3337 subclassing of the interface.

3338 The following snippet shows an implementation of the HelloService marked with the @Service
3339 annotation.

```
3340 package services.hello;  
3341  
3342 import org.oasisopen.sca.annotation.Service;  
3343  
3344 @Service(HelloService.class)  
3345 public class HelloServiceImpl implements HelloService {  
3346  
3347     public void hello(String name) {  
3348         System.out.println("Hello " + name);  
3349     }  
3350 }  
3351
```

3352

11 WSDL to Java and Java to WSDL

3353 This specification applies the WSDL to Java and Java to WSDL mapping rules as defined by [the](#)
 3354 [JAX-WS 2.1 specification \[JAX-WS\]](#) for generating remotable Java interfaces from WSDL portTypes
 3355 and vice versa.

3356 SCA runtimes MUST support the JAX-WS 2.1 mappings from WSDL to Java and from Java to
 3357 WSDL. [JCA100022] For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime
 3358 MUST treat a Java interface as if it had a @WebService annotation on the class, even if it doesn't.
 3359 [JCA100001] The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation
 3360 as a synonym for the @javax.jws.OneWay annotation. [JCA100002] For the WSDL-to-Java
 3361 mapping, the SCA runtime MUST take the generated @WebService annotation to imply that the
 3362 Java interface is @Remotable. [JCA100003]

3363 For the mapping from Java types to XML schema types, SCA permits both the JAXB 2.1 [JAX-B]
 3364 mapping and the SDO 2.1 [SDO] mapping. SCA runtimes MUST support the JAXB 2.1 mapping
 3365 from XML Schema to Java and from Java to XML Schema. [JCA100004] SCA runtimes MAY support
 3366 the SDO 2.1 mapping from XML schema types to Java and from Java to XML Schema.
 3367 [JCA100005] Having a choice of binding technologies is allowed, as noted in the first paragraph of
 3368 section 5 of the JSR 181 (version 2) specification, which is referenced by the JAX-WS specification.

11.1 JAX-WS Annotations and SCA Interfaces

3370 A Java class or interface used to define an SCA interface can contain JAX-WS annotations. In
 3371 addition to affecting the Java to WSDL mapping defined by [the JAX-WS specification \[JAX-WS\]](#)
 3372 these annotations can impact the SCA interface. An SCA runtime MUST apply the JAX-WS
 3373 annotations as described in Table 11-1 and Table 11-2 when introspecting a Java class or interface
 3374 class. [JCA100011] This could mean that the interface of a Java implementation is defined by a
 3375 WSDL interface declaration.

Annotation	Property	Impact to SCA Interface
@WebService		A Java interface or class annotated with @WebService MUST be treated as if annotated with the SCA @Remotable annotation [JCA100012]
	name	If used to define a service, sets service name
	targetNamespace	None
	serviceName	None
	wsdlLocation	A Java class annotated with the @WebService annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition instead of the annotated Java class. [JCA100013]
	endpointInterface	A Java class annotated with the @WebService annotation with its endpointInterface attribute set MUST have its interface defined by the referenced interface instead of annotated Java class. [JCA100014]
	portName	None
@WebMethod		
	operationName	Sets operation name

	action	None
	exclude	Method is excluded from the interface.
@OneWay		The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation. [JCA100002]
@WebParam		
	name	Sets parameter name
	targetNamespace	None
	mode	Sets directionality of parameter
	header	A Java class or interface containing an @WebParam annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface. [JCA100015]
	partName	Overrides name
@WebResult		
	name	Sets parameter name
	targetNamespace	None
	header	A Java class or interface containing an @WebResult annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface. [JCA100016]
	partName	Overrides name
@SOAPBinding		A Java class or interface containing an @SOAPBinding annotation MUST be treated as if the SOAP intent is applied to the Java class or interface. [JCA100021]
	style	
	use	
	parameterStyle	
@HandlerChain		None
	file	
	name	

3376 Table 11-1: JSR 181 Annotations and SCA Interfaces

3377

Annotation	Property	Impact to SCA Interface
@ServiceMode		A Java class containing an @ServiceMode annotation MUST be treated as if the SOAP intent is applied to the Java class. [JCA100017]

<i>Annotation</i>	<i>Property</i>	<i>Impact to SCA Interface</i>
	value	
@WebFault		
	name	Sets fault name
	targetNamespace	None
	faultBean	None
@RequestWrapper		None
	localName	
	targetNamespace	
	className	
@ResponseWrapper		None
	localName	
	targetNamespace	
	className	
@WebServiceClient		An interface or class annotated with @WebServiceClient MUST NOT be used to define an SCA interface. [JCA100018]
	name	
	targetNamespace	
	wsdlLocation	
@WebEndpoint		None
	name	
@WebServiceProvider		A class annotated with @WebServiceProvider MUST be treated as if annotated with the SCA @Remotable annotation. [JCA100019]
	wsdlLocation	A Java class annotated with the @WebServiceProvider annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition is used instead of the annotated Java class. [JCA100020]
	serviceName	None
	portName	None
	targetNamespace	None
@BindingType		None
	value	
@WebServiceRef		See JEE specification

<i>Annotation</i>	<i>Property</i>	<i>Impact to SCA Interface</i>
	name	
	wSDLLocation	
	type	
	value	
	mappedName	
@WebServiceRefs		See JEE specification
	value	
@Action		None
	fault	
	input	
	output	
@FaultAction		None
	value	
	output	

3378 *Table 11-2: JSR 224 Annotations and SCA Interfaces*

3379

3380 11.2 JAX-WS Client Asynchronous API for a Synchronous Service

3381 The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a
3382 client application with a means of invoking that service asynchronously, so that the client can
3383 invoke a service operation and proceed to do other work without waiting for the service operation
3384 to complete its processing. The client application can retrieve the results of the service either
3385 through a polling mechanism or via a callback method which is invoked when the operation
3386 completes.

3387 For SCA service interfaces defined using interface.java, the Java interface MUST NOT contain the
3388 additional client-side asynchronous polling and callback methods defined by JAX-WS. [JCA100006]
3389 For SCA reference interfaces defined using interface.java, the SCA runtime MUST support a Java
3390 interface which contains the additional client-side asynchronous polling and callback methods
3391 defined by JAX-WS. [JCA100007] If the additional client-side asynchronous polling and callback
3392 methods defined by JAX-WS are present in the interface which declares the type of a reference in
3393 the implementation, SCA Runtimes MUST NOT include these methods in the SCA reference
3394 interface in the component type of the implementation. [JCA100008]
3395

3396 The additional client-side asynchronous polling and callback methods defined by JAX-WS are
3397 recognized in a Java interface as follows:

3398 For each method M in the interface, if another method P in the interface has

- 3399 a. a method name that is M's method name with the characters "Async" appended, and
- 3400 b. the same parameter signature as M, and
- 3401 c. a return type of Response<R> where R is the return type of M

3402 then P is a JAX-WS polling method that isn't part of the SCA interface contract.

3403 For each method M in the interface, if another method C in the interface has
3404 a. a method name that is M's method name with the characters "Async" appended, and
3405 b. a parameter signature that is M's parameter signature with an additional final parameter of
3406 type AsyncHandler<R> where R is the return type of M, and
3407 c. a return type of Future<?>
3408 then C is a JAX-WS callback method that isn't part of the SCA interface contract.
3409 As an example, an interface can be defined in WSDL as follows:

```
3410 <!-- WSDL extract -->  
3411 <message name="getPrice">  
3412 <part name="ticker" type="xsd:string"/>  
3413 </message>  
3414  
3415 <message name="getPriceResponse">  
3416 <part name="price" type="xsd:float"/>  
3417 </message>  
3418  
3419 <portType name="StockQuote">  
3420 <operation name="getPrice">  
3421 <input message="tns:getPrice"/>  
3422 <output message="tns:getPriceResponse"/>  
3423 </operation>  
3424 </portType>
```

3425
3426 The JAX-WS asynchronous mapping will produce the following Java interface:

```
3427 // asynchronous mapping  
3428 @WebService  
3429 public interface StockQuote {  
3430 float getPrice(String ticker);  
3431 Response<Float> getPriceAsync(String ticker);  
3432 Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);  
3433 }
```

3434
3435 For SCA interface definition purposes, this is treated as equivalent to the following:

```
3436 // synchronous mapping  
3437 @WebService  
3438 public interface StockQuote {  
3439 float getPrice(String ticker);  
3440 }
```

3441
3442 **SCA runtimes MUST support the use of the JAX-WS client asynchronous model.** [JCA100009] In
3443 the above example, if the client implementation uses the asynchronous form of the interface, the
3444 two additional getPriceAsync() methods can be used for polling and callbacks as defined by the
3445 JAX-WS specification.

3446 11.3 Treatment of SCA Asynchronous Service API

3447 **For SCA service interfaces defined using interface.java, the SCA runtime MUST support a Java**
3448 **interface which contains the server-side asynchronous methods defined by SCA.** [JCA100010]

3449 Asynchronous service methods are identified as described in the section "Asynchronous handling
3450 of Long Running Service Operations" and are mapped to WSDL in the same way as the equivalent
3451 synchronous method described in that section.

3452 Generating an asynchronous service method from a WSDL request/response operation follows the
3453 algorithm described in the same section.

3454 12 Conformance

3455 The XML schema pointed to by the RDDDL document at the namespace URI, defined by this
3456 specification, are considered to be authoritative and take precedence over the XML schema
3457 defined in the appendix of this document.

3458 Normative code artifacts related to this specification are considered to be authoritative and take
3459 precedence over specification text.

3460 There are three categories of artifacts for which this specification defines conformance:

- 3461 a) SCA Java XML Document,
- 3462 b) SCA Java Class
- 3463 c) SCA Runtime.

3464 12.1 SCA Java XML Document

3465 An SCA Java XML document is an SCA Composite Document, or an SCA ComponentType
3466 Document, as defined by the [SCA Assembly Model specification \[ASSEMBLY\]](#), that uses the
3467 <interface.java> element. Such an SCA Java XML document MUST be a conformant SCA
3468 Composite Document or SCA ComponentType Document, as defined by the [SCA Assembly Model
3469 specification \[ASSEMBLY\]](#), and MUST comply with the requirements specified in [the Interface
3470 section](#) of this specification.

3471 12.2 SCA Java Class

3472 An SCA Java Class is a Java class or interface that complies with Java Standard Edition version 5.0
3473 and MAY include annotations and APIs defined in this specification. An SCA Java Class that uses
3474 annotations and APIs defined in this specification MUST comply with the requirements specified in
3475 this specification for those annotations and APIs.

3476 12.3 SCA Runtime

3477 The APIs and annotations defined in this specification are meant to be used by Java-based
3478 component implementation models in either partial or complete fashion. A Java-based component
3479 implementation specification that uses this specification specifies which of the APIs and
3480 annotations defined here are used. The APIs and annotations an SCA Runtime has to support
3481 depends on which Java-based component implementation specification the runtime supports. For
3482 example, see the [SCA POJO Component Implementation Specification \[JAVA_CI\]](#).

3483 An implementation that claims to conform to this specification MUST meet the following
3484 conditions:

- 3485 1. The implementation MUST meet all the conformance requirements defined by the SCA Assembly
3486 Model Specification [ASSEMBLY].
- 3487 2. The implementation MUST support <interface.java> and MUST comply with all the normative
3488 statements in Section 3.
- 3489 3. The implementation MUST reject an SCA Java XML Document that does not conform to the sca-
3490 interface-java.xsd schema.
- 3491 4. The implementation MUST support and comply with all the normative statements in Section 10.

3492

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

```
3493 <?xml version="1.0" encoding="UTF-8"?>
3494 <!-- Copyright(C) OASIS(R) 2005,2010. All Rights Reserved.
3495 OASIS trademark, IPR and other policies apply. -->
3496 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3497 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
3498 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
3499 elementFormDefault="qualified">
3500
3501 <include schemaLocation="sca-core-1.1-cd04.xsd"/>
3502
3503 <!-- Java Interface -->
3504 <element name="interface.java" type="sca:JavaInterface"
3505 substitutionGroup="sca:interface"/>
3506 <complexType name="JavaInterface">
3507 <complexContent>
3508 <extension base="sca:Interface">
3509 <sequence>
3510 <any namespace="##other" processContents="lax" minOccurs="0"
3511 maxOccurs="unbounded"/>
3512 </sequence>
3513 <attribute name="interface" type="NCName" use="required"/>
3514 <attribute name="callbackInterface" type="NCName"
3515 use="optional"/>
3516 <attribute name="remotable" type="boolean" use="optional"/>
3517 </extension>
3518 </complexContent>
3519 </complexType>
3520
3521 </schema>
3522
```

3523

B. Java Classes and Interfaces

3524

B.1 SCAClient Classes and Interfaces

3525

B.1.1 SCAClientFactory Class

3526 SCA provides an abstract base class SCAClientFactory. Vendors can provide subclasses of this class
3527 which create objects that implement the SCAClientFactory class suitable for linking to services in their
3528 SCA runtime.

3529

3530

```
/*  
 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.  
 * OASIS trademark, IPR and other policies apply.  
 */
```

3533

```
package org.oasisopen.sca.client;
```

3535

3536

```
import java.net.URI;
```

3537

```
import java.util.Properties;
```

3538

3539

```
import org.oasisopen.sca.NoSuchDomainException;
```

3540

```
import org.oasisopen.sca.NoSuchServiceException;
```

3541

```
import org.oasisopen.sca.client.SCAClientFactoryFinder;
```

3542

```
import org.oasisopen.sca.client.impl.SCAClientFactoryFinderImpl;
```

3543

3544

```
/**  
 * The SCAClientFactory can be used by non-SCA managed code to  
 * lookup services that exist in a SCADomain.  
 *  
 * @see SCAClientFactoryFinderImpl  
 * @see SCAClient  
 *  
 * @author OASIS Open  
 */
```

3552

3553

```
public abstract class SCAClientFactory {
```

3555

3556

```
    /**  
     * The SCAClientFactoryFinder.  
     * Provides a means by which a provider of an SCAClientFactory  
     * implementation can inject a factory finder implementation into  
     * the abstract SCAClientFactory class - once this is done, future  
     * invocations of the SCAClientFactory use the injected factory  
     * finder to locate and return an instance of a subclass of  
     * SCAClientFactory.  
     */
```

3564

```
    protected static SCAClientFactoryFinder factoryFinder;
```

3566

```
    /**  
     * The Domain URI of the SCA Domain which is accessed by this  
     * SCAClientFactory  
     */
```

3569

```
    private URI domainURI;
```

3570

3571

```
    /**  
     * Prevent concrete subclasses from using the no-arg constructor
```

3572

3573

```

3574     */
3575     private SCAClientFactory() {
3576     }
3577
3578     /**
3579     * Constructor used by concrete subclasses
3580     * @param domainURI - The Domain URI of the Domain accessed via this
3581     * SCAClientFactory
3582     */
3583     protected SCAClientFactory(URI domainURI) {
3584         throws NoSuchDomainException {
3585             this.domainURI = domainURI;
3586         }
3587
3588     /**
3589     * Gets the Domain URI of the Domain accessed via this SCAClientFactory
3590     * @return - the URI for the Domain
3591     */
3592     protected URI getDomainURI() {
3593         return domainURI;
3594     }
3595
3596
3597     /**
3598     * Creates a new instance of the SCAClient that can be
3599     * used to lookup SCA Services.
3600     *
3601     * @param domainURI      URI of the target domain for the SCAClient
3602     * @return A new SCAClient
3603     */
3604     public static SCAClientFactory newInstance( URI domainURI )
3605         throws NoSuchDomainException {
3606         return newInstance( null, null, domainURI);
3607     }
3608
3609     /**
3610     * Creates a new instance of the SCAClient that can be
3611     * used to lookup SCA Services.
3612     *
3613     * @param properties     Properties that may be used when
3614     * creating a new instance of the SCAClient
3615     * @param domainURI      URI of the target domain for the SCAClient
3616     * @return A new SCAClient instance
3617     */
3618     public static SCAClientFactory newInstance(Properties properties,
3619                                               URI domainURI)
3620         throws NoSuchDomainException {
3621         return newInstance(properties, null, domainURI);
3622     }
3623
3624     /**
3625     * Creates a new instance of the SCAClient that can be
3626     * used to lookup SCA Services.
3627     *
3628     * @param classLoader    ClassLoader that may be used when
3629     * creating a new instance of the SCAClient
3630     * @param domainURI      URI of the target domain for the SCAClient
3631     * @return A new SCAClient instance

```

```

3632     */
3633     public static SCAClientFactory newInstance(ClassLoader classLoader,
3634                                               URI domainURI)
3635         throws NoSuchDomainException {
3636         return newInstance(null, classLoader, domainURI);
3637     }
3638
3639     /**
3640     * Creates a new instance of the SCAClient that can be
3641     * used to lookup SCA Services.
3642     *
3643     * @param properties    Properties that may be used when
3644     * creating a new instance of the SCAClient
3645     * @param classLoader   ClassLoader that may be used when
3646     * creating a new instance of the SCAClient
3647     * @param domainURI     URI of the target domain for the SCAClient
3648     * @return A new SCAClient instance
3649     */
3650     public static SCAClientFactory newInstance(Properties properties,
3651                                               ClassLoader classLoader,
3652                                               URI domainURI)
3653         throws NoSuchDomainException {
3654         final SCAClientFactoryFinder finder =
3655             factoryFinder != null ? factoryFinder :
3656             new SCAClientFactoryFinderImpl();
3657         final SCAClientFactory factory
3658             = finder.find(properties, classLoader, domainURI);
3659         return factory;
3660     }
3661
3662     /**
3663     * Returns a reference proxy that implements the business interface <T>
3664     * of a service in the SCA Domain handled by this SCAClientFactory
3665     *
3666     * @param serviceURI the relative URI of the target service. Takes the
3667     * form componentName/serviceName.
3668     * Can also take the extended form componentName/serviceName/bindingName
3669     * to use a specific binding of the target service
3670     *
3671     * @param interfaze The business interface class of the service in the
3672     * domain
3673     * @param <T> The business interface class of the service in the domain
3674     *
3675     * @return a proxy to the target service, in the specified SCA Domain
3676     * that implements the business interface <B>.
3677     * @throws NoSuchServiceException Service requested was not found
3678     * @throws NoSuchDomainException Domain requested was not found
3679     */
3680     public abstract <T> T getService(Class<T> interfaze, String serviceURI)
3681         throws NoSuchServiceException, NoSuchDomainException;
3682 }

```

3683 B.1.2 SCAClientFactoryFinder interface

3684 The SCAClientFactoryFinder interface is a Service Provider Interface representing a SCAClientFactory
3685 finder. SCA provides a default reference implementation of this interface. SCA runtime vendors can
3686 create alternative implementations of this interface that use different class loading or lookup mechanisms.
3687

```

3688  /*
3689  * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
3690  * OASIS trademark, IPR and other policies apply.
3691  */
3692
3693  package org.oasisopen.sca.client;
3694
3695  import java.net.URI;
3696  import java.util.Properties;
3697
3698  import org.oasisopen.sca.NoSuchDomainException;
3699
3700  /* A Service Provider Interface representing a SCAClientFactory finder.
3701  * SCA provides a default reference implementation of this interface.
3702  * SCA runtime vendors can create alternative implementations of this
3703  * interface that use different class loading or lookup mechanisms.
3704  */
3705  public interface SCAClientFactoryFinder {
3706
3707      /**
3708       * Method for finding the SCAClientFactory for a given Domain URI using
3709       * a specified set of properties and a a specified ClassLoader
3710       * @param properties - properties to use - may be null
3711       * @param classLoader - ClassLoader to use - may be null
3712       * @param domainURI - the Domain URI - must be a valid SCA Domain URI
3713       * @return - the SCAClientFactory or null if the factory could not be
3714       * @throws - NoSuchDomainException if the domainURI does not reference
3715       * a valid SCA Domain
3716       * found
3717       */
3718      SCAClientFactory find(Properties properties,
3719                          ClassLoader classLoader,
3720                          URI domainURI )
3721          throws NoSuchDomainException ;
3722  }

```

3723 B.1.3 SCAClientFactoryFinderImpl class

3724 This class provides a default implementation for finding a provider's SCAClientFactory implementation
3725 class. It is used if the provider does not inject its SCAClientFactoryFinder implementation class into the
3726 base SCAClientFactory class.

3727 It discovers a provider's SCAClientFactory implementation by referring to the following information in this
3728 order:

- 3729 1. The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on the
3730 newInstance() method call if specified
- 3731 2. The org.oasisopen.sca.client.SCAClientFactory property from the System Properties
- 3732 3. The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file

```

3733  /*
3734  * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
3735  * OASIS trademark, IPR and other policies apply.
3736  */
3737  package org.oasisopen.sca.client.impl;
3738
3739  import org.oasisopen.sca.client.SCAClientFactoryFinder;
3740
3741  import java.io.BufferedReader;

```

```

3742 import java.io.Closeable;
3743 import java.io.IOException;
3744 import java.io.InputStream;
3745 import java.io.InputStreamReader;
3746 import java.lang.reflect.Constructor;
3747 import java.net.URI;
3748 import java.net.URL;
3749 import java.util.Properties;
3750
3751 import org.oasisopen.sca.NoSuchDomainException;
3752 import org.oasisopen.sca.ServiceRuntimeException;
3753 import org.oasisopen.sca.client.SCAClientFactory;
3754
3755 /**
3756  * This is a default implementation of an SCAClientFactoryFinder which is
3757  * used to find an implementation of the SCAClientFactory interface.
3758  *
3759  * @see SCAClientFactoryFinder
3760  * @see SCAClientFactory
3761  *
3762  * @author OASIS Open
3763  */
3764 public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder {
3765
3766     /**
3767      * The name of the System Property used to determine the SPI
3768      * implementation to use for the SCAClientFactory.
3769      */
3770     private static final String SCA_CLIENT_FACTORY_PROVIDER_KEY =
3771         SCAClientFactory.class.getName();
3772
3773     /**
3774      * The name of the file loaded from the ClassPath to determine
3775      * the SPI implementation to use for the SCAClientFactory.
3776      */
3777     private static final String SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE
3778         = "META-INF/services/" + SCA_CLIENT_FACTORY_PROVIDER_KEY;
3779
3780     /**
3781      * Public Constructor
3782      */
3783     public SCAClientFactoryFinderImpl() {
3784     }
3785
3786     /**
3787      * Creates an instance of the SCAClientFactorySPI implementation.
3788      * This discovers the SCAClientFactorySPI Implementation and instantiates
3789      * the provider's implementation.
3790      *
3791      * @param properties Properties that may be used when creating a new
3792      * instance of the SCAClient
3793      * @param classLoader ClassLoader that may be used when creating a new
3794      * instance of the SCAClient
3795      * @return new instance of the SCAClientFactory
3796      * @throws ServiceRuntimeException Failed to create SCAClientFactory
3797      * Implementation.
3798      */
3799     public SCAClientFactory find(Properties properties,

```

```

3800         ClassLoader classLoader,
3801         URI domainURI )
3802     throws NoSuchDomainException, ServiceRuntimeException {
3803         if (classLoader == null) {
3804             classLoader = getThreadContextClassLoader ();
3805         }
3806         final String factoryImplClassName =
3807             discoverProviderFactoryImplClass(properties, classLoader);
3808         final Class<? extends SCAClientFactory> factoryImplClass
3809             = loadProviderFactoryClass(factoryImplClassName,
3810                                     classLoader);
3811         final SCAClientFactory factory =
3812             instantiateSCAClientFactoryClass(factoryImplClass,
3813                                             domainURI );
3814         return factory;
3815     }
3816
3817 /**
3818  * Gets the Context ClassLoader for the current Thread.
3819  *
3820  * @return The Context ClassLoader for the current Thread.
3821  */
3822 private static ClassLoader getThreadContextClassLoader () {
3823     final ClassLoader threadClassLoader =
3824         Thread.currentThread().getContextClassLoader();
3825     return threadClassLoader;
3826 }
3827
3828 /**
3829  * Attempts to discover the class name for the SCAClientFactorySPI
3830  * implementation from the specified Properties, the System Properties
3831  * or the specified ClassLoader.
3832  *
3833  * @return The class name of the SCAClientFactorySPI implementation
3834  * @throw ServiceRuntimeException Failed to find implementation for
3835  * SCAClientFactorySPI.
3836  */
3837 private static String
3838     discoverProviderFactoryImplClass(Properties properties,
3839                                     ClassLoader classLoader)
3840     throws ServiceRuntimeException {
3841     String providerClassName =
3842         checkPropertiesForSPIClassName(properties);
3843     if (providerClassName != null) {
3844         return providerClassName;
3845     }
3846
3847     providerClassName =
3848         checkPropertiesForSPIClassName(System.getProperties());
3849     if (providerClassName != null) {
3850         return providerClassName;
3851     }
3852
3853     providerClassName = checkMETAINFServicesForSIPClassName(classLoader);
3854     if (providerClassName == null) {
3855         throw new ServiceRuntimeException(
3856             "Failed to find implementation for SCAClientFactory");
3857     }

```

```

3858
3859     return providerClassName;
3860 }
3861
3862 /**
3863  * Attempts to find the class name for the SCAClientFactorySPI
3864  * implementation from the specified Properties.
3865  *
3866  * @return The class name for the SCAClientFactorySPI implementation
3867  * or <code>null</code> if not found.
3868  */
3869 private static String
3870     checkPropertiesForSPIClassName(Properties properties) {
3871     if (properties == null) {
3872         return null;
3873     }
3874
3875     final String providerClassName =
3876         properties.getProperty(SCA_CLIENT_FACTORY_PROVIDER_KEY);
3877     if (providerClassName != null && providerClassName.length() > 0) {
3878         return providerClassName;
3879     }
3880
3881     return null;
3882 }
3883
3884 /**
3885  * Attempts to find the class name for the SCAClientFactorySPI
3886  * implementation from the META-INF/services directory
3887  *
3888  * @return The class name for the SCAClientFactorySPI implementation or
3889  * <code>null</code> if not found.
3890  */
3891 private static String checkMETAINFServicesForSIPClassName(ClassLoader cl)
3892 {
3893     final URL url =
3894         cl.getResource(SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE);
3895     if (url == null) {
3896         return null;
3897     }
3898
3899     InputStream in = null;
3900     try {
3901         in = url.openStream();
3902         BufferedReader reader = null;
3903         try {
3904             reader =
3905                 new BufferedReader(new InputStreamReader(in, "UTF-8"));
3906
3907             String line;
3908             while ((line = readNextLine(reader)) != null) {
3909                 if (!line.startsWith("#") && line.length() > 0) {
3910                     return line;
3911                 }
3912             }
3913
3914             return null;
3915         } finally {

```

```

3916         closeStream(reader);
3917     }
3918 } catch (IOException ex) {
3919     throw new ServiceRuntimeException(
3920         "Failed to discover SCAClientFactory provider", ex);
3921 } finally {
3922     closeStream(in);
3923 }
3924 }
3925
3926 /**
3927  * Reads the next line from the reader and returns the trimmed version
3928  * of that line
3929  *
3930  * @param reader The reader from which to read the next line
3931  * @return The trimmed next line or <code>null</code> if the end of the
3932  * stream has been reached
3933  * @throws IOException I/O error occurred while reading from Reader
3934  */
3935 private static String readNextLine(BufferedReader reader)
3936     throws IOException {
3937
3938     String line = reader.readLine();
3939     if (line != null) {
3940         line = line.trim();
3941     }
3942     return line;
3943 }
3944
3945 /**
3946  * Loads the specified SCAClientFactory Implementation class.
3947  *
3948  * @param factoryImplClassName The name of the SCAClientFactory
3949  * Implementation class to load
3950  * @return The specified SCAClientFactory Implementation class
3951  * @throws ServiceRuntimeException Failed to load the SCAClientFactory
3952  * Implementation class
3953  */
3954 private static Class<? extends SCAClientFactory>
3955     loadProviderFactoryClass(String factoryImplClassName,
3956         ClassLoader classLoader)
3957     throws ServiceRuntimeException {
3958
3959     try {
3960         final Class<?> providerClass =
3961             classLoader.loadClass(factoryImplClassName);
3962         final Class<? extends SCAClientFactory> providerFactoryClass =
3963             providerClass.asSubclass(SCAClientFactory.class);
3964         return providerFactoryClass;
3965     } catch (ClassNotFoundException ex) {
3966         throw new ServiceRuntimeException(
3967             "Failed to load SCAClientFactory implementation class "
3968             + factoryImplClassName, ex);
3969     } catch (ClassCastException ex) {
3970         throw new ServiceRuntimeException(
3971             "Loaded SCAClientFactory implementation class "
3972             + factoryImplClassName
3973             + " is not a subclass of "

```

```

3974         + SCAClientFactory.class.getName() , ex);
3975     }
3976 }
3977
3978 /**
3979  * Instantiate an instance of the specified SCAClientFactorySPI
3980  * Implementation class.
3981  *
3982  * @param factoryImplClass The SCAClientFactorySPI Implementation
3983  * class to instantiate.
3984  * @return An instance of the SCAClientFactorySPI Implementation class
3985  * @throws ServiceRuntimeException Failed to instantiate the specified
3986  * specified SCAClientFactorySPI Implementation class
3987  */
3988 private static SCAClientFactory instantiateSCAClientFactoryClass(
3989     Class<? extends SCAClientFactory> factoryImplClass,
3990     URI domainURI)
3991     throws NoSuchDomainException, ServiceRuntimeException {
3992
3993     try {
3994         Constructor<? extends SCAClientFactory> URIConstructor =
3995             factoryImplClass.getConstructor(domainURI.getClass());
3996         SCAClientFactory provider =
3997             URIConstructor.newInstance( domainURI );
3998         return provider;
3999     } catch (Throwable ex) {
4000         throw new ServiceRuntimeException(
4001             "Failed to instantiate SCAClientFactory implementation class "
4002             + factoryImplClass, ex);
4003     }
4004 }
4005
4006 /**
4007  * Utility method for closing Closeable Object.
4008  *
4009  * @param closeable The Object to close.
4010  */
4011 private static void closeStream(Closeable closeable) {
4012     if (closeable != null) {
4013         try{
4014             closeable.close();
4015         } catch (IOException ex) {
4016             throw new ServiceRuntimeException("Failed to close stream",
4017                 ex);
4018         }
4019     }
4020 }
4021 }

```

4022 **B.1.4 SCAClient Classes and Interfaces - what does a vendor need to do?**

4023 The SCAClient classes and interfaces are designed so that vendors can provide their own
4024 implementation suited to the needs of their SCA runtime. This section describes the tasks that a vendor
4025 needs to consider in relation to the SCAClient classes and interfaces.

- 4026 • Implement their SCAClientFactory implementation class

4027 Vendors need to provide a subclass of SCAClientFactory that is capable of looking up Services in
4028

4029 their SCA Runtime. Vendors need to subclass SCAClientFactory and implement the getService()
4030 method so that it creates reference proxies to services in SCA Domains handled by their SCA
4031 runtime(s).

4032

4033

4034 • Configure the Vendor SCAClientFactory implementation class so that it gets used
4035 Vendors have several options:

4036

4037 Option 1: Set System Property to point to the Vendor's implementation

4038

4039 Vendors set the org.oasisopen.sca.client.SCAClientFactory System Property to point to their
4040 implementation class and use the reference implementation of SCAClientFactoryFinder

4041

4042 Option 2: Provide a META-INF/services file

4043

4044 Vendors provide a META-INF/services/org.oasisopen.sca.client.SCAClientFactory file that points
4045 to their implementation class and use the reference implementation of SCAClientFactoryFinder

4046

4047 Option 3: Inject a vendor implementation of the SCAClientFactoryFinder interface into
4048 SCAClientFactory

4049

4050 Vendors inject an instance of the vendor implementation of SCAClientFactoryFinder into the
4051 factoryFinder field of the SCAClientFactory abstract class. The reference implementation of
4052 SCAClientFactoryFinder is not used in this scenario. The vendor implementation of
4053 SCAClientFactoryFinder can find the vendor implementation(s) of SCAClientFactory by any
4054 means.

4055

4056

4057

C. Conformance Items

4058 This section contains a list of conformance items for the SCA-J Common Annotations and APIs
4059 specification.
4060

Conformance ID	Description
[JCA20001]	Remotable Services MUST NOT make use of <i>method overloading</i> .
[JCA20002]	the SCA runtime MUST ensure that a stateless scoped implementation instance object is only ever dispatched on one thread at any one time.
[JCA20003]	within the SCA lifecycle of a stateless scoped implementation instance, the SCA runtime MUST only make a single invocation of one business method.
[JCA20004]	Where an implementation is used by a "domain level component", and the implementation is marked "Composite" scope, the SCA runtime MUST ensure that all consumers of the component appear to be interacting with a single runtime instance of the implementation.
[JCA20005]	When the implementation class is marked for eager initialization, the SCA runtime MUST create a composite scoped instance when its containing component is started.
[JCA20006]	If a method of an implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when the implementation instance is created.
[JCA20007]	the SCA runtime MAY run multiple threads in a single composite scoped implementation instance object and the SCA runtime MUST NOT perform any synchronization.
[JCA20008]	Where an implementation is marked "Composite" scope and it is used by a component that is nested inside a composite that is used as the implementation of a higher level component, the SCA runtime MUST ensure that all consumers of the component appear to be interacting with a single runtime instance of the implementation. There can be multiple instances of the higher level component, each running on different nodes in a distributed SCA runtime.
[JCA20009]	The SCA runtime MAY use by-reference semantics when passing input parameters, return values or exceptions on calls to remotable services within the same JVM if both the service method implementation and the service proxy used by the client are marked "allows pass by reference".
[JCA20010]	The SCA runtime MUST use by-value semantics when passing input parameters, return values and exceptions on calls to remotable services within the same JVM if the service method implementation is not marked "allows pass by reference" or the service proxy used by the client is not marked "allows pass by reference".
[JCA30001]	The value of the @interface attribute MUST be the fully qualified name of the Java interface class
[JCA30002]	The value of the @callbackInterface attribute MUST be the fully

- qualified name of a Java interface used for callbacks
- [JCA30003] if the Java interface class identified by the @interface attribute does contain a Java @Callback annotation, then the Java interface class identified by the @callbackInterface attribute MUST be the same interface class.
- [JCA30004] The interface.java element MUST conform to the schema defined in the sca-interface-java.xsd schema.
- [JCA30005] The value of the @remotable attribute on the <interface.java/> element does not override the presence of a @Remotable annotation on the interface class and so if the interface class contains a @Remotable annotation and the @remotable attribute has a value of "false", then the SCA Runtime MUST raise an error and MUST NOT run the component concerned.
- [JCA30006] A Java interface referenced by the @interface attribute of an <interface.java/> element MUST NOT contain any of the following SCA Java annotations:
@AllowsPassByReference, @ComponentName, @Constructor, @Context, @Destroy, @EagerInit, @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service.
- [JCA30007] A Java interface referenced by the @callbackInterface attribute of an <interface.java/> element MUST NOT contain any of the following SCA Java annotations:
@AllowsPassByReference, @Callback, @ComponentName, @Constructor, @Context, @Destroy, @EagerInit, @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service.
- [JCA30009] The SCA Assembly Model specification [ASSEMBLY] defines a number of criteria that need to be satisfied in order for two interfaces to be compatible or have a compatible superset or subset relationship. If these interfaces are both Java interfaces, compatibility also means that every method that is present in both interfaces is defined consistently in both interfaces with respect to the @OneWay annotation, that is, the annotation is either present in both interfaces or absent in both interfaces.
- [JCA30010] If the identified class is annotated with either the JAX-WS @WebService or @WebServiceProvider annotations and the annotation has a non-empty **wsdlLocation** property, then the SCA Runtime MUST act as if an <interface.wsdl/> element is present instead of the <interface.java/> element, with an @interface attribute identifying the portType mapped from the Java interface class and containing @requires and @policySets attribute values equal to the @requires and @policySets attribute values of the <interface.java/> element.
- [JCA40001] The SCA Runtime MUST call a constructor of the component implementation at the start of the Constructing state.
- [JCA40002] The SCA Runtime MUST perform any constructor reference or property injection when it calls the constructor of a component implementation.
- [JCA40003] When the constructor completes successfully, the SCA Runtime MUST transition the component implementation to the Injecting state.

- [JCA40004] If an exception is thrown whilst in the Constructing state, the SCA Runtime MUST transition the component implementation to the Terminated state.
- [JCA40005] When a component implementation instance is in the Injecting state, the SCA Runtime MUST first inject all field and setter properties that are present into the component implementation.
- [JCA40006] When a component implementation instance is in the Injecting state, the SCA Runtime MUST inject all field and setter references that are present into the component implementation, after all the properties have been injected.
- [JCA40007] The SCA Runtime MUST ensure that the correct synchronization model is used so that all injected properties and references are made visible to the component implementation without requiring the component implementation developer to do any specific synchronization.
- [JCA40008] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation is in the Injecting state.
- [JCA40009] When the injection of properties and references completes successfully, the SCA Runtime MUST transition the component implementation to the Initializing state.
- [JCA40010] If an exception is thrown whilst injecting properties or references, the SCA Runtime MUST transition the component implementation to the Destroying state.
- [JCA40011] When the component implementation enters the Initializing State, the SCA Runtime MUST call the method annotated with @Init on the component implementation, if present.
- [JCA40012] If a component implementation invokes an operation on an injected reference that refers to a target that has not yet been initialized, the SCA Runtime MUST throw a ServiceUnavailableException.
- [JCA40013] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Initializing state.
- [JCA40014] Once the method annotated with @Init completes successfully, the SCA Runtime MUST transition the component implementation to the Running state.
- [JCA40015] If an exception is thrown whilst initializing, the SCA Runtime MUST transition the component implementation to the Destroying state.
- [JCA40016] The SCA Runtime MUST invoke Service methods on a component implementation instance when the component implementation is in the Running state and a client invokes operations on a service offered by the component.
- [JCA40017] When the component implementation scope ends, the SCA Runtime MUST transition the component implementation to the Destroying state.
- [JCA40018] When a component implementation enters the Destroying state, the SCA Runtime MUST call the method annotated with @Destroy on the

- component implementation, if present.
- [JCA40019] If a component implementation invokes an operation on an injected reference that refers to a target that has been destroyed, the SCA Runtime MUST throw an InvalidServiceException.
- [JCA40020] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Destroying state.
- [JCA40021] Once the method annotated with @Destroy completes successfully, the SCA Runtime MUST transition the component implementation to the Terminated state.
- [JCA40022] If an exception is thrown whilst destroying, the SCA Runtime MUST transition the component implementation to the Terminated state.
- [JCA40023] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Terminated state.
- [JCA40024] If a property or reference is unable to be injected, the SCA Runtime MUST transition the component implementation to the Destroying state.
- [JCA60001] When a bidirectional service is invoked, the SCA runtime MUST inject a callback reference for the invoking service into all fields and setter methods of the service implementation class that are marked with a @Callback annotation and typed by the callback interface of the bidirectional service, and the SCA runtime MUST inject null into all other fields and setter methods of the service implementation class that are marked with a @Callback annotation.
- [JCA60002] When a non-bidirectional service is invoked, the SCA runtime MUST inject null into all fields and setter methods of the service implementation class that are marked with a @Callback annotation.
- [JCA60003] The SCA asynchronous service Java interface mapping of a WSDL request-response operation MUST appear as follows:
The interface is annotated with the "asyncInvocation" intent.
For each service operation in the WSDL, the Java interface contains an operation with
- a name which is the JAX-WS mapping of the WSDL operation name, with the suffix "Async" added
 - a void return type
 - a set of input parameter(s) which match the JAX-WS mapping of the input parameter(s) of the WSDL operation plus an additional last parameter which is a ResponseDispatch object typed by the JAX-WS Response Bean mapping of the output parameter(s) of the WSDL operation, where ResponseDispatch is the type defined in the SCA Java Common Annotations and APIs specification.
- [JCA60004] An SCA Runtime MUST support the use of the SCA asynchronous service interface for the interface of an SCA service.
- [JCA60005] If the SCA asynchronous service interface ResponseDispatch handleResponse method is invoked more than once through either its sendResponse or its sendFault method, the SCA runtime MUST throw

an `IllegalStateException`.

[JCA60006]

For the purposes of matching interfaces (when wiring between a reference and a service, or when using an implementation class by a component), an interface which has one or more methods which follow the SCA asynchronous service pattern MUST be treated as if those methods are mapped as the equivalent synchronous methods, as follows:

Asynchronous service methods are characterized by:

- f) void return type
- g) a method name with the suffix "Async"
- h) a last input parameter with a type of `ResponseDispatch<X>`
- i) annotation with the `asyncInvocation` intent
- j) possible annotation with the `@AsyncFault` annotation

The mapping of each such method is as if the method had the return type "X", the method name without the suffix "Async" and all the input parameters except the last parameter of the type `ResponseDispatch<X>`, plus the list of exceptions contained in the `@AsyncFault` annotation.

[JCA70001]

SCA identifies annotations that correspond to intents by providing an `@Intent` annotation which MUST be used in the definition of a specific intent annotation.

[JCA70002]

Intent annotations MUST NOT be applied to the following:

A method of a service implementation class, except for a setter method that is either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification

A service implementation class field that is not either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification

A service implementation class constructor parameter that is not annotated with `@Reference`

[JCA70003]

Where multiple intent annotations (general or specific) are applied to the same Java element, the SCA runtime MUST compute the combined intents for the Java element by merging the intents from all intent annotations on the Java element according to the SCA Policy Framework [POLICY] rules for merging intents at the same hierarchy level.

[JCA70004]

If intent annotations are specified on both an interface method and the method's declaring interface, the SCA runtime MUST compute the effective intents for the method by merging the combined intents from the method with the combined intents for the interface according to the SCA Policy Framework [POLICY] rules for merging intents within a structural hierarchy, with the method at the lower level and the interface at the higher level.

[JCA70005]

The `@PolicySets` annotation MUST NOT be applied to the following:

A method of a service implementation class, except for a setter method that is either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component

Implementation specification

A service implementation class field that is not either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification

A service implementation class constructor parameter that is not annotated with `@Reference`

- [JCA70006] If the `@PolicySets` annotation is specified on both an interface method and the method's declaring interface, the SCA runtime MUST compute the effective policy sets for the method by merging the policy sets from the method with the policy sets from the interface.
- [JCA80001] The `ComponentContext.getService` method MUST throw an `IllegalArgumentException` if the reference identified by the `referenceName` parameter has multiplicity of `0..n` or `1..n`.
- [JCA80002] The `ComponentContext.getRequestContext` method MUST return non-null when invoked during the execution of a Java business method for a service operation or a callback operation, on the same thread that the SCA runtime provided, and MUST return null in all other cases.
- [JCA80003] When invoked during the execution of a service operation, the `RequestContext.getServiceReference` method MUST return a `ServiceReference` that represents the service that was invoked.
- [JCA80004] The `ComponentContext.getServiceReference` method MUST throw an `IllegalArgumentException` if the reference named by the `referenceName` parameter has multiplicity greater than one.
- [JCA80005] The `ComponentContext.getServiceReference` method MUST throw an `IllegalArgumentException` if the reference named by the `referenceName` parameter does not have an interface of the type defined by the `businessInterface` parameter.
- [JCA80006] The `ComponentContext.getServiceReference` method MUST throw an `IllegalArgumentException` if the component does not have a reference with the name provided in the `referenceName` parameter.
- [JCA80007][JCA80007] The `ComponentContext.getServiceReference` method MUST return null if the multiplicity of the reference named by the `referenceName` parameter is `0..1` and the reference has no target service configured.
- [JCA80008] The `ComponentContext.getURI` method MUST return the absolute URI of the component in the SCA Domain.
- [JCA80009] The `ComponentContext.getService` method MUST return the proxy object implementing the interface provided by the `businessInterface` parameter, for the reference named by the `referenceName` parameter with the interface defined by the `businessInterface` parameter when that reference has a target service configured.
- [JCA80010] The `ComponentContext.getService` method MUST return null if the multiplicity of the reference named by the `referenceName` parameter is `0..1` and the reference has no target service configured.
- [JCA80011] The `ComponentContext.getService` method MUST throw an

- IllegalArgumentException if the component does not have a reference with the name supplied in the referenceName parameter.
- [JCA80012] The ComponentContext.getService method MUST throw an IllegalArgumentException if the service reference with the name supplied in the referenceName does not have an interface compatible with the interface supplied in the businessInterface parameter.
- [JCA80013] The ComponentContext.getServiceReference method MUST return a ServiceReference object typed by the interface provided by the businessInterface parameter, for the reference named by the referenceName parameter with the interface defined by the businessInterface parameter when that reference has a target service configured.
- [JCA80014] The ComponentContext.getServices method MUST return a collection containing one proxy object implementing the interface provided by the businessInterface parameter for each of the target services configured on the reference identified by the referenceName parameter.
- [JCA80015] The ComponentContext.getServices method MUST return an empty collection if the service reference with the name supplied in the referenceName parameter is not wired to any target services.
- [JCA80016] The ComponentContext.getServices method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..1 or 1..1.
- [JCA80017] The ComponentContext.getServices method MUST throw an IllegalArgumentException if the component does not have a reference with the name supplied in the referenceName parameter.
- The ComponentContext.getServices method MUST throw an IllegalArgumentException if the service reference with the name supplied in the referenceName does not have an interface compatible with the interface supplied in the businessInterface parameter.[JCA80018]
- [JCA80019] The ComponentContext.getServiceReferences method MUST return a collection containing one ServiceReference object typed by the interface provided by the businessInterface parameter for each of the target services configured on the reference identified by the referenceName parameter.
- [JCA80020] The ComponentContext.getServiceReferences method MUST return an empty collection if the service reference with the name supplied in the referenceName parameter is not wired to any target services.
- [JCA80021] The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..1 or 1..1.
- [JCA80022] The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if the component does not have a reference

with the name supplied in the `referenceName` parameter.

- [JCA80023] The `ComponentContext.getServiceReferences` method MUST throw an `IllegalArgumentException` if the service reference with the name supplied in the `referenceName` does not have an interface compatible with the interface supplied in the `businessInterface` parameter.
- [JCA80024] The `ComponentContext.createSelfReference` method MUST return a `ServiceReference` object typed by the interface defined by the `businessInterface` parameter for one of the services of the invoking component which has the interface defined by the `businessInterface` parameter.
- [JCA80025] The `ComponentContext.getServiceReferences` method MUST throw an `IllegalArgumentException` if the component does not have a service which implements the interface identified by the `businessInterface` parameter.
- [JCA80026] The `ComponentContext.createSelfReference` method MUST return a `ServiceReference` object typed by the interface defined by the `businessInterface` parameter for the service identified by the `serviceName` of the invoking component and which has the interface defined by the `businessInterface` parameter.
- [JCA80027] The `ComponentContext.createSelfReference` method MUST throw an `IllegalArgumentException` if the component does not have a service with the name identified by the `serviceName` parameter.
- [JCA80028] The `ComponentContext.createSelfReference` method MUST throw an `IllegalArgumentException` if the component service with the name identified by the `serviceName` parameter does not implement a business interface which is compatible with the supplied `businessInterface` parameter.
- [JCA80029] The `ComponentContext.getProperty` method MUST return an object of the type identified by the `type` parameter containing the value specified in the component configuration for the property named by the `propertyName` parameter or null if no value is specified in the configuration.
- [JCA80030] The `ComponentContext.getProperty` method MUST throw an `IllegalArgumentException` if the component does not have a property with the name identified by the `propertyName` parameter.
- [JCA80031] The `ComponentContext.getProperty` method MUST throw an `IllegalArgumentException` if the component property with the name identified by the `propertyName` parameter does not have a type which is compatible with the supplied `type` parameter.
- [JCA80032] The `ComponentContext.cast` method MUST return a `ServiceReference` object which is typed by the same business interface as specified by the reference proxy object supplied in the `target` parameter.
- [JCA80033] The `ComponentContext.cast` method MUST throw an `IllegalArgumentException` if the supplied `target` parameter is not an SCA reference proxy object.
- [JCA80034] The `RequestContext.getSecuritySubject` method MUST return the JAAS subject of the current request, or null if there is no subject or null if the method is invoked from code not processing a service request or

callback request.

- [JCA80035] The RequestContext.getServiceName method MUST return the name of the service for which an operation is being processed, or null if invoked from a thread that is not processing a service operation or a callback operation.
- [JCA80036] The RequestContext.getCallbackReference method MUST return a ServiceReference object typed by the interface of the callback supplied by the client of the invoked service, or null if either the invoked service is not bidirectional or if the method is invoked from a thread that is not processing a service operation.
- [JCA80037] The RequestContext.getCallback method MUST return a reference proxy object typed by the interface of the callback supplied by the client of the invoked service, or null if either the invoked service is not bidirectional or if the method is invoked from a thread that is not processing a service operation.
- [JCA80038] When invoked during the execution of a callback operation, the RequestContext.getServiceReference method MUST return a ServiceReference that represents the callback that was invoked.
- [JCA80039] When invoked from a thread not involved in the execution of either a service operation or of a callback operation, the RequestContext.getServiceReference method MUST return null.
- [JCA80040] The ServiceReference.getService method MUST return a reference proxy object which can be used to invoke operations on the target service of the reference and which is typed with the business interface of the reference.
- [JCA80041] The ServiceReference.getBusinessInterface method MUST return a Class object representing the business interface of the reference.
- [JCA80042] The SCAClientFactory.newInstance(URI) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.
- [JCA80043] The SCAClientFactory.newInstance(URI) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.
- [JCA80044] The SCAClientFactory.newInstance(Properties, URI) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.
- [JCA80045] The SCAClientFactory.newInstance(Properties, URI) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.
- [JCA80046] The SCAClientFactory.newInstance(Classloader, URI) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.
- [JCA80047] The SCAClientFactory.newInstance(Classloader, URI) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.
- [JCA80048] The SCAClientFactory.newInstance(Properties, Classloader, URI) method MUST return an object which implements the

- SCAClientFactory class for the SCA Domain identified by the domainURI parameter.
- [JCA80049] The SCAClientFactory.newInstance(Properties, Classloader, URI) MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.
- [JCA80050] The SCAClientFactory.getService method MUST return a proxy object which implements the business interface defined by the interfaze parameter and which can be used to invoke operations on the service identified by the serviceURI parameter.
- [JCA80051] The SCAClientFactory.getService method MUST throw a NoSuchServiceException if a service with the relative URI serviceURI and a business interface which matches interfaze cannot be found in the SCA Domain targeted by the SCAClient object.
- [JCA80052] The SCAClientFactory.getService method MUST throw a NoSuchServiceException if the domainURI of the SCAClientFactory does not identify a valid SCA Domain.
- [JCA80053] The SCAClientFactory.getDomainURI method MUST return the SCA Domain URI of the Domain associated with the SCAClientFactory object.
- [JCA80054] The SCAClientFactory.getDomainURI method MUST throw a **NoSuchServiceException** if the domainURI of the SCAClientFactory does not identify a valid SCA Domain.
- The implementation of the SCAClientFactoryFinder.find method MUST return an object which is an implementation of the SCAClientFactory interface, for the SCA Domain represented by the doaminURI parameter, using the supplied properties and classloader.
- [JCA80055] The implementation of the SCAClientFactoryFinder.find method MUST return an object which is an implementation of the SCAClientFactory interface, for the SCA Domain represented by the doaminURI parameter, using the supplied properties and classloader.
- [JCA80056] The implementation of the SCAClientFactoryFinder.find method MUST throw a ServiceRuntimeException if the SCAClientFactory implementation could not be found.
- [JCA50057] The ResponseDispatch.sendResponse() method MUST send the response message to the client of an asynchronous service.
- [JCA80058] The ResponseDispatch.sendResponse() method MUST throw an InvalidStateException if either the sendResponse method or the sendFault method has already been called once.
- [JCA80059] The ResponseDispatch.sendFault() method MUST send the supplied fault to the client of an asynchronous service.
- [JCA50060] The ResponseDispatch.sendFault() method MUST throw an InvalidStateException if either the sendResponse method or the sendFault method has already been called once.
- [JCA90001] An SCA runtime MUST verify the proper use of all SCA annotations and if an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the invalid implementation code.

- [JCA90002] SCA annotations MUST NOT be used on static methods or on 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.
- [JCA90003] If a constructor of an implementation class is annotated with @Constructor and the constructor has parameters, each of these parameters MUST have either a @Property annotation or a @Reference annotation.
- [JCA90004] A method annotated with @Destroy can have any access modifier and MUST have a void return type and no arguments.
- [JCA90005] If there is a method annotated with @Destroy that matches the criteria for the annotation, the SCA runtime MUST call the annotated method when the scope defined for the implementation class ends.
- [JCA90007] When marked for eager initialization with an @EagerInit annotation, the composite scoped instance MUST be created when its containing component is started.
- [JCA90008] A method marked with the @Init annotation can have any access modifier and MUST have a void return type and no arguments.
- [JCA90009] If there is a method annotated with @Init that matches the criteria for the annotation, the SCA runtime MUST call the annotated method after all property and reference injection is complete.
- [JCA90011] The @Property annotation MUST NOT be used on a class field that is declared as final.
- [JCA90013] For a @Property annotation applied to a constructor parameter, there is no default value for the name attribute and the name attribute MUST be present.
- [JCA90014] For a @Property annotation applied to a constructor parameter, the required attribute MUST NOT have the value false.
- [JCA90015] The @Qualifier annotation MUST be used in a specific intent annotation definition where the intent has qualifiers.
- [JCA90016] The @Reference annotation MUST NOT be used on a class field that is declared as final.
- [JCA90018] For a @Reference annotation applied to a constructor parameter, there is no default for the name attribute and the name attribute MUST be present.
- [JCA90019] For a @Reference annotation applied to a constructor parameter, the required attribute MUST have the value true.
- [JCA90020] If the type of a reference is not an array or any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity= 0..1 if the @Reference annotation required attribute is false and with @multiplicity=1..1 if the @Reference annotation required attribute is true.
- [JCA90021] If the type of a reference is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a

	<reference/> element with @multiplicity=0..n if the @Reference annotation required attribute is false and with @multiplicity=1..n if the @Reference annotation required attribute is true.
[JCA90022]	An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by the SCA runtime as null (either via injection or via API call).
[JCA90023]	An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty collection (either via injection or via API call).
[JCA90024]	References MAY be reinjected by an SCA runtime after the initial creation of a component if the reference target changes due to a change in wiring that has occurred since the component was initialized.
[JCA90025]	In order for reinjection to occur, the following MUST be true: <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.
[JCA90026]	If a reference target changes and the reference is not reinjected, the reference MUST continue to work as if the reference target was not changed.
[JCA90027]	If an operation is called on a reference where the target of that reference has been undeployed, the SCA runtime SHOULD throw an InvalidServiceException.
[JCA90028]	If an operation is called on a reference where the target of the reference has become unavailable for some reason, the SCA runtime SHOULD throw a ServiceUnavailableException.
[JCA90029]	If the target service of the reference is changed, the reference MUST either continue to work or throw an InvalidServiceException when it is invoked.
[JCA90030]	A ServiceReference that has been obtained from a reference by ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast(). If the reference is subsequently reinjected, the ServiceReference obtained from the original reference MUST continue to work as if the reference target was not changed.
[JCA90031]	If the target of a ServiceReference has been undeployed, the SCA runtime SHOULD throw a InvalidServiceException when an operation is invoked on the ServiceReference.
[JCA90032]	If the target of a ServiceReference has become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an operation is invoked on the ServiceReference.
[JCA90033]	If the target service of a ServiceReference is changed, the reference MUST either continue to work or throw an InvalidServiceException when it is invoked.
[JCA90034]	A reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() MUST correspond to the current configuration of the domain. This applies whether or not reinjection has taken place.

- [JCA90035] If the target of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service, and attempts to call business methods SHOULD throw an InvalidServiceException or a ServiceUnavailableException.
- [JCA90036] If the target service of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has changed, the returned value SHOULD be a reference to the changed service.
- [JCA90037] in the cases where reference reinjection is not allowed, the array or Collection for a reference of multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes occur to the reference wiring or to the targets of the wiring.
- [JCA90038] In cases where the contents of a reference array or collection change when the wiring changes or the targets change, then for references that use setter injection, the setter method MUST be called by the SCA runtime for any change to the contents.
- [JCA90039] A reinjected array or Collection for a reference MUST NOT be the same array or Collection object previously injected to the component.
- [JCA90040] A remotable service can be published externally as a service and MUST be translatable into a WSDL portType.
- [JCA90041] The @Scope annotation MUST only be used on a service's implementation class. It is an error to use this annotation on an interface.
- [JCA90042] An implementation class need not be declared as implementing all of the interfaces implied by the services declared in its @Service annotation, but all methods of all the declared service interfaces MUST be present.
- [JCA90045] If a component implementation has two services with the same Java simple name, the names attribute of the @Service annotation MUST be specified.
- [JCA90046] When used to annotate a method or a field of an implementation class for injection of a callback object, the @Callback annotation MUST NOT specify any attributes.
- [JCA90047] For a @Property annotation, if the type of the Java class field or the type of the input parameter of the setter method or constructor is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <property/> element with a @many attribute set to true, otherwise @many MUST be set to false.
- [JCA90050] The number of Strings in the names attribute array of the @Service annotation MUST match the number of elements in the value attribute array.
- [JCA90052] The @AllowsPassByReference annotation MUST only annotate the following locations:
a service implementation class

- an individual method of a remotable service implementation
- an individual reference which uses a remotable interface, where the reference is a field, a setter method, or a constructor parameter
- [JCA90053] The `@Remotable` annotation is valid only on a Java interface, a Java class, a field, a setter method, or a constructor parameter. It MUST NOT appear anywhere else.
- [JCA90054] When used to annotate a method or a field of an implementation class for injection of a callback object, the type of the method or field MUST be the callback interface of at least one bidirectional service offered by the implementation class.
- [JCA90055] A method annotated with `@OneWay` MUST have a void return type and MUST NOT have declared checked exceptions.
- [JCA90056] When a method of a Java interface is annotated with `@OneWay`, the SCA runtime MUST ensure that all invocations of that method are executed in a non-blocking fashion, as described in the section on Asynchronous Programming.
- [JCA90057] The `@Callback` annotation MUST NOT appear on a setter method or a field of a Java implementation class that has COMPOSITE scope.
- [JCA90058] When used to annotate a setter method or a field of an implementation class for injection of a callback object, the SCA runtime MUST inject a callback reference proxy into that method or field when the Java class is initialized, if the component is invoked via a service which has a callback interface and where the type of the setter method or field corresponds to the type of the callback interface.
- [JCA90060] The value of each element in the `@Service` names array MUST be unique amongst all the other element values in the array.
- [JCA90061] When the Java type of a field, setter method or constructor parameter with the `@Property` annotation is a primitive type or a JAXB annotated class, the SCA runtime MUST convert a property value specified by an SCA component definition into an instance of the Java type as defined by the XML to Java mapping in the JAXB specification [JAXB] with XML schema validation enabled.
- [JCA100001] For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java interface as if it had a `@WebService` annotation on the class, even if it doesn't.
- [JCA100002] The SCA runtime MUST treat an `@org.oasisopen.sca.annotation.OneWay` annotation as a synonym for the `@javax.jws.OneWay` annotation.
- [JCA100003] For the WSDL-to-Java mapping, the SCA runtime MUST take the generated `@WebService` annotation to imply that the Java interface is `@Remotable`.
- [JCA100004] SCA runtimes MUST support the JAXB 2.1 mapping from XML Schema to Java and from Java to XML Schema.
- [JCA100005] SCA runtimes MAY support the SDO 2.1 mapping from XML schema types to Java and from Java to XML Schema.
- [JCA100006] For SCA service interfaces defined using `interface.java`, the Java interface MUST NOT contain the additional client-side asynchronous

- polling and callback methods defined by JAX-WS.
- [JCA100007] For SCA reference interfaces defined using interface.java, the SCA runtime MUST support a Java interface which contains the additional client-side asynchronous polling and callback methods defined by JAX-WS.
- [JCA100008] If the additional client-side asynchronous polling and callback methods defined by JAX-WS are present in the interface which declares the type of a reference in the implementation, SCA Runtimes MUST NOT include these methods in the SCA reference interface in the component type of the implementation.
- [JCA100009] SCA runtimes MUST support the use of the JAX-WS client asynchronous model.
- [JCA100010] For SCA service interfaces defined using interface.java, the SCA runtime MUST support a Java interface which contains the server-side asynchronous methods defined by SCA.
- [JCA100011] An SCA runtime MUST apply the JAX-WS annotations as described in Table 11-1 and Table 11-2 when introspecting a Java class or interface class.
- [JCA100012] A Java interface or class annotated with @WebService MUST be treated as if annotated with the SCA @Remotable annotation
- [JCA100013] A Java class annotated with the @WebService annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition instead of the annotated Java class.
- [JCA100014] A Java class annotated with the @WebService annotation with its endpointInterface attribute set MUST have its interface defined by the referenced interface instead of annotated Java class.
- [JCA100015] A Java class or interface containing an @WebParam annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface.
- [JCA100016] A Java class or interface containing an @WebResult annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface.
- [JCA100017] A Java class containing an @ServiceMode annotation MUST be treated as if the SOAP intent is applied to the Java class.
- [JCA100018] An interface or class annotated with @WebServiceClient MUST NOT be used to define an SCA interface.
- [JCA100019] A class annotated with @WebServiceProvider MUST be treated as if annotated with the SCA @Remotable annotation.
- [JCA100020] A Java class annotated with the @WebServiceProvider annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition is used instead of the annotated Java class.
- [JCA100021] A Java class or interface containing an @SOAPBinding annotation MUST be treated as if the SOAP intent is applied to the Java class or interface.

[JCA100022]

SCA runtimes MUST support the JAX-WS 2.1 mappings from WSDL to Java and from Java to WSDL.

4061

4062

D. Acknowledgements

4063 The following individuals have participated in the creation of this specification and are gratefully
4064 acknowledged:

4065 **Participants:**

Participant Name	Affiliation
Bryan Aupperle	IBM
Ron Barack	SAP AG*
Michael Beisiegel	IBM
Henning Blohm	SAP AG*
David Booz	IBM
Martin Chapman	Oracle Corporation
Graham Charters	IBM
Shih-Chang Chen	Oracle Corporation
Chris Cheng	Primeton Technologies, Inc.
Vamsavardhana Reddy Chillakuru	IBM
Roberto Chinnici	Sun Microsystems
Pyounguk Cho	Oracle Corporation
Eric Clairambault	IBM
Mark Combella	Avaya, Inc.
Jean-Sebastien Delfino	IBM
Mike Edwards	IBM
Raymond Feng	IBM
Bo Ji	Primeton Technologies, Inc.
Uday Joshi	Oracle Corporation
Anish Karmarkar	Oracle Corporation
Michael Keith	Oracle Corporation
Rainer Kerth	SAP AG*
Meeraj Kunnumpurath	Individual
Simon Laws	IBM
Yang Lei	IBM
Mark Little	Red Hat
Ashok Malhotra	Oracle Corporation
Jim Marino	Individual
Jeff Mischkinsky	Oracle Corporation
Sriram Narasimhan	TIBCO Software Inc.
Simon Nash	Individual
Sanjay Patil	SAP AG*
Plamen Pavlov	SAP AG*
Peter Peshev	SAP AG*
Ramkumar Ramalingam	IBM
Luciano Resende	IBM
Michael Rowley	Active Endpoints, Inc.
Vladimir Savchenko	SAP AG*
Pradeep Simha	TIBCO Software Inc.
Raghav Srinivasan	Oracle Corporation

Scott Vorthmann
Feng Wang
Robin Yang

TIBCO Software Inc.
Primeton Technologies, Inc.
Primeton Technologies, Inc.

4066

4067

E. Non-Normative Text

4069

F. Revision History

4070 [optional; should not be included in OASIS Standards]

4071

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
cd01-rev4	2009-01-18	Anish Karmarkar	* Applied resolutions of issues 28, 52, 94, 96, 99, 101
cd02	2009-01-26	Mike Edwards	Minor editorial cleanup. All changes accepted.

			All comments removed.
cd02-rev1	2009-02-03	Mike Edwards	Issues 25+95 Issue 120
cd02-rev2	2009-02-08	Mike Edwards	Merge annotation definitions contained in section 10 into section 8 Move remaining parts of section 10 to section 7. Accept all changes.
cd02-rev3	2009-03-16	Mike Edwards	Issue 104 - RFC2119 work and formal marking of all normative statements - all sections - Completion of Appendix B (list of all normative statements) Accept all changes
cd02-rev4	2009-03-20	Mike Edwards	Editorially removed sentence about componentType side files in Section 1 Editorially changed package name to org.oasisopen from org.osoa in lines 291, 292 Issue 6 - add Section 2.3, modify section 9.1 Issue 30 - Section 2.2.2 Issue 76 - Section 6.2.4 Issue 27 - Section 7.6.2, 7.6.2.1 Issue 77 - Section 1.2 Issue 102 - Section 9.21 Issue 123 - conersations removed Issue 65 - Added a new Section 4 ** Causes renumbering of later sections ** ** NB new numbering is used below ** Issue 119 - Added a new section 12 Issue 125 - Section 3.1 Issue 130 - (new number) Section 8.6.2.1 Issue 132 - Section 1 Issue 133 - Section 10.15, Section 10.17 Issue 134 - Section 10.3, Section 10.18 Issue 135 - Section 10.21 Issue 138 - Section 11 Issue 141 - Section 9.1 Issue 142 - Section 10.17.1
cd02-rev5	2009-04-20	Mike Edwards	Issue 154 - Appendix A Issue 129 - Section 8.3.1.1
cd02-rev6	2009-04-28	Mike Edwards	Issue 148 - Section 3 Issue 98 - Section 8
cd02-rev7	2009-04-30	Mike Edwards	Editorial cleanup throughout the spec

cd02-rev8	2009-05-01	Mike Edwards	Further extensive editorial cleanup throughout the spec Issue 160 - Section 8.6.2 & 8.6.2.1 removed
cd02-rev8a	2009-05-03	Simon Nash	Minor editorial cleanup
cd03	2009-05-04	Anish Karmarkar	Updated references and front page clean up
cd03-rev1	2009-09-15	David Booz	Applied Issues: 1,13,125,131,156,157,158,159,161,165,172,177
cd03-rev2	2010-01-19	David Booz	Updated to current Assembly namespace Applied issues: 127,155,168,181,184,185,187,189,190,194
cd03-rev3	2010-02-01	Mike Edwards	Applied issue 54. Editorial updates to code samples.

4072