



Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

**Committee Draft 02, Revision 01+AnnotationsMerge
03 February 2009**

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev1.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev1.doc>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev1.pdf> (normative)

Previous Version:

Latest Version:

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

Latest Approved Version:

Technical Committee:

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

Chair(s):

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

Editor(s):

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

Related work:

This specification replaces or supersedes:

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

This specification is related to:

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

Declared XML Namespace(s):

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

Abstract:

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

Specifically, this specification covers:

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

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

Status:

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

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

For information on whether any patents have been disclosed that 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, 2008. All Rights Reserved.

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

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

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

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

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

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

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

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

Table of Contents

1	Introduction	6
1.1	Terminology	6
1.2	Normative References.....	6
1.3	Non-Normative References.....	7
2	Implementation Metadata	8
2.1	Service Metadata	8
2.1.1	@Service	8
2.1.2	Java Semantics of a Remotable Service.....	8
2.1.3	Java Semantics of a Local Service	8
2.1.4	@Reference.....	9
2.1.5	@Property.....	9
2.2	Implementation Scopes: @Scope, @Init, @Destroy	9
2.2.1	Stateless scope.....	9
2.2.2	Composite scope	10
3	Interface.....	10
3.1	Java interface element – <interface.java>	12
3.2	@Remotable	13
3.3	@Callback.....	13
4	Client API.....	14
4.1	Accessing Services from an SCA Component	14
4.1.1	Using the Component Context API	14
4.2	Accessing Services from non-SCA component implementations	14
4.2.1	ComponentContext.....	14
5	Error Handling.....	15
6	Asynchronous Programming.....	16
6.1	@OneWay	16
6.2	Callbacks	16
6.2.1	Using Callbacks	16
6.2.2	Callback Instance Management.....	18
6.2.3	Implementing Multiple Bidirectional Interfaces	18
6.2.4	Accessing Callbacks	19
7	Policy Annotations for Java.....	20
7.1	General Intent Annotations.....	20
7.2	Specific Intent Annotations.....	22
7.2.1	How to Create Specific Intent Annotations	22
7.3	Application of Intent Annotations	23
7.3.1	Inheritance And Annotation.....	23
7.4	Relationship of Declarative And Annotated Intents.....	25
7.5	Policy Set Annotations	25
7.6	Security Policy Annotations.....	26
7.6.1	Security Interaction Policy.....	26
7.6.2	Security Implementation Policy	27
8	Java API	30

8.1	Component Context	30
8.2	Request Context	31
8.3	ServiceReference	32
8.4	ServiceRuntimeException	32
8.5	ServiceUnavailableException	33
8.6	InvalidServiceException	33
8.7	Constants Interface	33
9	Java Annotations	34
9.1	@AllowsPassByReference	34
9.2	@Authentication	35
9.3	@Callback	36
9.4	@ComponentName	37
9.5	@Confidentiality	38
9.6	@Constructor	38
9.7	@Context	39
9.8	@Destroy	40
9.9	@EagerInit	40
9.10	@Init	41
9.11	@Integrity	41
9.12	@Intent	42
9.13	@OneWay	43
9.14	@PolicySet	43
9.15	@Property	44
9.16	@Qualifier	45
9.17	@Reference	46
	9.17.1 Reinjection	48
9.18	@Remotable	49
9.19	@Requires	51
9.20	@Scope	51
9.21	@Service	52
10	WSDL to Java and Java to WSDL	54
	10.1 JAX-WS Client Asynchronous API for a Synchronous Service	54
A.	XML Schema: sca-interface-java.xsd	56
B.	Conformance Items	57
C.	Acknowledgements	58
D.	Non-Normative Text	59
E.	Revision History	60

1 Introduction

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

Specifically, this specification covers:

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

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

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

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

1.1 Terminology

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

1.2 Normative References

- | | |
|------------|---|
| [RFC2119] | S. Bradner, <i>Key words for use in RFCs to Indicate Requirement Levels</i> , http://www.ietf.org/rfc/rfc2119.txt , IETF RFC 2119, March 1997. |
| [ASSEMBLY] | SCA Assembly Specification, http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf |
| [SDO] | SDO 2.1 Specification, http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf |
| [JAX-B] | JAXB 2.1 Specification, http://www.jcp.org/en/jsr/detail?id=222 |
| [WSDL] | WSDL Specification,
WSDL 1.1: http://www.w3.org/TR/wsdl ,
WSDL 2.0: http://www.w3.org/TR/wsdl20/ |
| [POLICY] | SCA Policy Framework, http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf |

- 44 **[JSR-250]** Common Annotation for Java Platform specification (JSR-250),
45 <http://www.jcp.org/en/jsr/detail?id=250>
46 **[JAX-WS]** JAX-WS 2.1 Specification (JSR-224),
47 <http://www.jcp.org/en/jsr/detail?id=224>
48 **[JAVABEANS]** JavaBeans 1.01 Specification,
49 <http://java.sun.com/javase/technologies/desktop/javabeans/api/>
50

51 **1.3 Non-Normative References**

- 52 **None** None

53 2 Implementation Metadata

54 This section describes SCA Java-based metadata, which applies to Java-based implementation
55 types.

56 2.1 Service Metadata

57 2.1.1 @Service

58
59 The **@Service annotation** is used on a Java class to specify the interfaces of the services
60 implemented by the implementation. Service interfaces are defined in one of the following ways:

- 61 • As a Java interface
- 62 • As a Java class
- 63 • As a Java interface generated from a Web Services Description Language [WSDL]
64 (WSDL) portType (Java interfaces generated from a WSDL portType are always
65 **remotable**)

66 2.1.2 Java Semantics of a Remotable Service

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

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

```
72 package services.hello;  
73 @Remotable  
74 public interface HelloService {  
75     String hello(String message);  
76 }  
77
```

78 2.1.3 Java Semantics of a Local Service

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

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

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

```
84 package services.hello;  
85 public interface HelloService {  
86     String hello(String message);  
87 }  
88
```

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

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

94 2.1.4 @Reference

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

98 2.1.5 @Property

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

102 2.2 Implementation Scopes: @Scope, @Init, @Destroy

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

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

109 This document defines two scopes:

- 110 • STATELESS
- 111 • COMPOSITE

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

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

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

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

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

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

```
124     @Init  
125     public void start() {  
126         ...  
127     }  
128  
129  
130     @Destroy  
131     public void stop() {  
132         ...  
133     }  
134
```

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

137 2.2.1 Stateless scope

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

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

146 2.2.2 Composite scope

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

150 A composite scoped implementation can also specify eager initialization using the **@EagerInit**
151 annotation. When marked for eager initialization, the composite scoped instance is created when
152 its containing component is started. If a method is marked with the @Init annotation, it is called
153 when the instance is created.

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

157 2.2.3 @AllowsPassByReference

158 Calls to remotable services (see section 3.2) have by-value semantics. This means that input
159 parameters passed to the service can be modified by the service without these modifications being
160 visible to the client. Similarly, the return value or exception from the service can be modified by
161 the client without these modifications being visible to the service implementation. For remote
162 calls (either cross-machine or cross-process), these semantics are a consequence of marshalling
163 input parameters, return values and exceptions "on the wire" and unmarshalling them "off the
164 wire" which results in physical copies being made. For local method calls within the same JVM,
165 Java language calling semantics are by-reference and therefore do not provide the correct by-
166 value semantics for SCA remotable interfaces. To compensate for this, the SCA runtime can
167 intervene in these calls to provide by-value semantics by making copies of any mutable objects
168 passed.

169 The cost of such copying can be very high relative to the cost of making a local call, especially if
170 the data being passed is large. Also, in many cases this copying is not needed if the
171 implementation observes certain conventions for how input parameters, return values and
172 exceptions are used. The @AllowsPassByReference annotation allows service method
173 implementations and client references to be marked as "allows pass by reference" to indicate that
174 they use input parameters, return values and exceptions in a manner that allows the SCA runtime
175 to avoid the cost of copying mutable objects when a remotable service is called locally within the
176 same JVM.

177 2.2.3.1 Marking services and references as "allows pass by reference"

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

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

185 See section 9.1 for details of how the @AllowsPassByReference annotation is used to mark a
186 service method implementation as "allows pass by reference".

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

Formatted: Bulleted + Level: 1 +
Aligned at: 38.75 pt + Indent at:
56.75 pt

- 189
- The client implementation will not modify any of the method's input parameters before the method returns. Such modifications might occur in callbacks or separate client threads.
 - If the method is one-way, the client implementation will not modify any of the method's input parameters at any time after calling the method. This is because one-way method calls return immediately without waiting for the service method to complete.
- 192
- 193
- 194
- 195 See section 9.1 for details of how the @AllowsPassByReference annotation is used to mark a client
- 196 reference as "allows pass by reference".

Formatted: Bulleted + Level: 1 +
Aligned at: 42.25 pt + Indent at:
60.25 pt

197 **2.2.3.2 Applying "allows pass by reference" to service proxies**

198 Service method calls are made by clients using service proxies, which can be obtained by injection

199 into client references or by making API calls. A service proxy is marked as "allows pass by

200 reference" if and only if any of the following applies:

- It is injected into a reference or callback reference that is marked "allows pass by reference".
- It is obtained by calling ComponentContext.getService() or ComponentContext.getServices() with the name of a reference that is marked "allows pass by reference".
- It is obtained by calling RequestContext.getCallback() from a service implementation that is marked "allows pass by reference".
- It is obtained by calling ServiceReference.getService() on a service reference that is marked "allows pass by reference" (see definition below).

Formatted: Bulleted + Level: 1 +
Aligned at: 38.3 pt + Indent at: 56.3
pt

210 A service reference for a remotable service call is marked "allows pass by reference" if and only if

211 any of the following applies:

- It is injected into a reference or callback reference that is marked "allows pass by reference".
- It is obtained by calling ComponentContext.getServiceReference() or ComponentContext.getServiceReferences() with the name of a reference that is marked "allows pass by reference".
- It is obtained by calling RequestContext.getCallbackReference() from a service implementation that is marked "allows pass by reference".
- It is obtained by calling ComponentContext.cast() on a proxy that is marked "allows pass by reference".

Formatted: Bulleted + Level: 1 +
Aligned at: 38.3 pt + Indent at: 56.3
pt

Formatted: Bulleted + Level: 1 +
Aligned at: 38.3 pt + Indent at: 56.3
pt

221 **2.2.3.3 Using "allows pass by reference" to optimize remotable calls**

222 The SCA runtime MAY use by-reference semantics when passing input parameters, return values

223 or exceptions on calls to remotable services within the same JVM if both the service method

224 implementation and the service proxy used by the client are marked "allows pass by reference".

225 The SCA runtime MUST use by-value semantics when passing input parameters, return values and

226 exceptions on calls to remotable services within the same JVM if the service method

227 implementation is not marked "allows pass by reference" or the service proxy used by the client is

228 not marked "allows pass by reference".

229 3 Interface

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

231 3.1 Java interface element – <interface.java>

232 The Java interface element is used in SCDL files in places where an interface is declared in terms
233 of a Java interface class. The Java interface element identifies the Java interface class and
234 optionally identifies a callback interface, where the first Java interface represents the forward
235 (service) call interface and the second interface represents the interface used to call back from the
236 service to the client.

237

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

239

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

241

242 The interface.java element has the following attributes:

- 243 • **interface (1..1)** – the Java interface class to use for the service interface. @interface MUST
244 be the fully qualified name of the Java interface class [JCA30001]
- 245 • **callbackInterface (0..1)** – the Java interface class to use for the callback interface.
246 @callbackInterface MUST be the fully qualified name of a Java interface used for callbacks
247 [JCA30002]

248

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

250

```
251 <interface.java interface="services.stockquote.StockQuoteService"  
252 callbackInterface="services.stockquote.StockQuoteServiceCallback" />
```

253

254 Here, the Java interface is defined in the Java class file

255 ./services/stockquote/StockQuoteService.class, where the root directory is defined by the
256 contribution in which the interface exists. Similarly, the callback interface is defined in the Java
257 class file ./services/stockquote/StockQuoteServiceCallback.class.

258 Note that the Java interface class identified by the @interface attribute can contain a Java
259 @Callback annotation which identifies a callback interface. If this is the case, then it is not
260 necessary to provide the @callbackInterface attribute. However, if the Java interface class
261 identified by the @interface attribute does contain a Java @Callback annotation, then the Java
262 interface class identified by the @callbackInterface attribute MUST be the same interface class.
263 [JCA30003]

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

268

269

270 **3.2 @Remotable**

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

Deleted: and

275 **3.3 @Callback**

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

281 4 Client API

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

284 4.1 Accessing Services from an SCA Component

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

289 4.1.1 Using the Component Context API

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

293 4.2 Accessing Services from non-SCA component implementations

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

296 4.2.1 ComponentContext

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

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

```
301  
302 ComponentContext context = // obtained via host environment-specific means  
303 HelloService helloService =  
304     context.getService(HelloService.class, "HelloService");  
305 String result = helloService.hello("Hello World!");
```

306 5 Error Handling

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

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

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

313 6 Asynchronous Programming

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

- 321 • support for non-blocking method calls
- 322 • callbacks

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

324 6.1 @OneWay

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

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

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

335 6.2 Callbacks

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

- 340 • an interface for the provided service
- 341 • a callback interface that must be provided by the client

342 Callbacks can be used for both remotable and local services. Either both interfaces of a
343 bidirectional service must be remotable, or both must be local. It is illegal to mix the two.

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

348 6.2.1 Using Callbacks

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

354 The following example shows a scenario in which bidirectional interfaces and callbacks could be
355 used. A client requests a quotation from a supplier. To process the enquiry and return the
356 quotation, some suppliers might need additional information from the client. The client does not

357 know which additional items of information will be needed by different suppliers. This interaction
358 can be modeled as a bidirectional interface with callback requests to obtain the additional
359 information.

```
360 package somepackage;  
361 import org.osoa.sca.annotation.Callback;  
362 import org.osoa.sca.annotation.Remotable;  
363 @Remotable  
364 @Callback(QuotationCallback.class)  
365 public interface Quotation {  
366     double requestQuotation(String productCode, int quantity);  
367 }  
368  
369 @Remotable  
370 public interface QuotationCallback {  
371     String getState();  
372     String getZipCode();  
373     String getCreditRating();  
374 }  
375
```

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

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

```
385 @Callback  
386 protected QuotationCallback callback;  
387  
388 public double requestQuotation(String productCode, int quantity) {  
389     double price = getPrice(productCode, quantity);  
390     double discount = 0;  
391     if (quantity > 1000 && callback.getState().equals("FL")) {  
392         discount = 0.05;  
393     }  
394     if (quantity > 10000 && callback.getCreditRating().charAt(0) == 'A') {  
395         discount += 0.05;  
396     }  
397     return price * (1-discount);  
398 }  
399  
400
```

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

```
404 public class ClientImpl implements ClientService, QuotationCallback {  
405  
406     private QuotationService myService;  
407  
408     @Reference  
409     public void setMyService(QuotationService service) {  
410         myService = service;  
411     }  
412
```

```

413
414     public void aClientMethod() {
415         ...
416         double quote = myService.requestQuotation("AB123", 2000);
417         ...
418     }
419
420     public String getState() {
421         return "TX";
422     }
423     public String getZipCode() {
424         return "78746";
425     }
426     public String getCreditRating() {
427         return "AA";
428     }
429 }

```

430 In this example the callback is *stateless*, i.e., the callback requests do not need any information
431 relating to the original service request. For a callback that needs information relating to the
432 original service request (a *stateful* callback), this information can be passed to the client by the
433 service provider as parameters on the callback request..
434

435 6.2.2 Callback Instance Management

436 Instance management for callback requests received by the client of the bidirectional service is
437 handled in the same way as instance management for regular service requests. If the client
438 implementation has STATELESS scope, the callback is dispatched using a newly initialized
439 instance. If the client implementation has COMPOSITE scope, the callback is dispatched using the
440 same shared instance that is used to dispatch regular service requests.

441 As described in section 6.7.1, a stateful callback can obtain information relating to the original
442 service request from parameters on the callback request. Alternatively, a composite-scoped client
443 could store information relating to the original request as instance data and retrieve it when the
444 callback request is received. These approaches could be combined by using a key passed on the
445 callback request (e.g., an order ID) to retrieve information that was stored in a composite-scoped
446 instance by the client code that made the original request.

447 6.2.3 Implementing Multiple Bidirectional Interfaces

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

```

454
455 @Callback
456 protected MyService1Callback callback1;
457
458 @Callback
459 protected MyService2Callback callback2;

```

460
461 If a single callback has a type that is compatible with multiple declared callback fields, then all of
462 them will be set.

463 6.2.4 Accessing Callbacks

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

467
468 A reference implementing the callback service interface can be obtained using
469 `ServiceReference.getService()`.

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

```
471 @Callback  
472 protected ServiceReference<MyCallback> callback;  
473  
474 public void someMethod() {  
475     MyCallback myCallback = callback.getCallback();    ...  
476  
477     myCallback.receiveResult(theResult);  
478 }  
479  
480  
481
```

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

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

```
488 public void someMethod() {  
489     MyCallback myCallback =  
490         ComponentContext.getRequestContext().getCallback();  
491     ...  
492     myCallback.receiveResult(theResult);  
493 }  
494  
495  
496  
497
```

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

```
500 @Context  
501 protected RequestContext requestContext;  
502  
503 void receiveResult(Object theResult) {  
504     Object refParams =  
505         requestContext.getServiceReference().getCallbackID();  
506     ...  
507 }  
508
```

509 This is necessary if the service implementation has COMPOSITE scope, because callback injection
510 is not performed for composite-scoped implementations.
511

512 7 Policy Annotations for Java

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

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

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

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

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

539

540 7.1 General Intent Annotations

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

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

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

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

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

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

```
553 public static final String SCA_PREFIX=  
554     "{http://docs.oasis-open.org/ns/opencsa/sca/200712";  
555 public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";  
556 public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";  
557
```

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

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

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

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

566

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

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

```
569     package com.foo;
570
571     import static org.oasisopen.sca.annotation.Confidentiality.*;
572     import static org.oasisopen.sca.annotation.Requires;
573     import static org.oasisopen.sca.annotation.Reference;
574
575     public class Foo {
576         @Requires(CONFIDENTIALITY)
577         @Reference
578         public void setBar(Bar bar) {
579             ...
580         }
581     }
582
```

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

```
586     package com.foo;
587
588     import static org.oasisopen.sca.Constants.*;
589     import static org.oasisopen.sca.annotation.Reference;
590     import static org.oasisopen.sca.annotation.Requires;
591
592     public class Foo {
593         @Requires(SCA_PREFIX+"confidentiality")
594         @Reference
595         public void setBar(Bar bar) {
596             ...
597         }
598     }
599
```

600 The formal syntax for the @Requires annotation follows:

```
601     @Requires( "qualifiedIntent" (, "qualifiedIntent")* )
```

602 where

```
603     qualifiedIntent ::= QName(.qualifier)*
```

604

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

606 7.2 Specific Intent Annotations

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

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

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

```
618     @Integrity
```

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

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

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

624 The general form of specific intent annotations is:

```
625     @<Intent>[(qualifiers)]
```

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

```
627     Intent      ::= NCName  
628     qualifiers  ::= "qualifier" (, "qualifier")*  
629     qualifier   ::= NCName(.qualifier)?  
630
```

631 7.2.1 How to Create Specific Intent Annotations

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

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

640 Alternatively, the `QName` of the intent can be specified using separate parameters for the
641 `targetNamespace` and the `localPart` for example:

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

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

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

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

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

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

653 See section @Qualifier for the formal definition of the @Qualifier annotation.

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

656 7.3 Application of Intent Annotations

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

- 658 • Java class
- 659 • Java interface
- 660 • Method
- 661 • Field
- 662 • Constructor parameter

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

```
665 @Authentication  
666 @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

667 In this case, the effective intents are "authentication", "confidentiality.message" and
668 "integrity.message".

669 If an annotation is specified at both the class/interface level and the method or field level, then
670 the method or field level annotation completely overrides the class level annotation of the same
671 base intent name.

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

675 7.3.1 Inheritance And Annotation

676 The inheritance rules for annotations are consistent with the common annotation specification, JSR
677 250.

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

```
680 package services.hello;  
681 import org.oasisopen.sca.annotation.Remotable;  
682 import org.oasisopen.sca.annotation.Integrity;  
683 import org.oasisopen.sca.annotation.Authentication;  
684  
685 @Integrity("transport")  
686 @Authentication  
687 public class HelloService {  
688     @Integrity  
689     @Authentication("message")  
690     public String hello(String message) {...}  
691  
692     @Integrity  
693     @Authentication("transport")  
694     public String helloThere() {...}  
695 }  
696  
697 package services.hello;  
698 import org.oasisopen.sca.annotation.Remotable;  
699 import org.oasisopen.sca.annotation.Confidentiality;  
700 import org.oasisopen.sca.annotation.Authentication;
```

```

701
702     @Confidentiality("message")
703     public class HelloChildService extends HelloService {
704         @Confidentiality("transport")
705         public String hello(String message) {...}
706         @Authentication
707         String helloWorld() {...}
708     }

```

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

710

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

713 The effective intent annotation on the hello method of the HelloChildService is
714 @Integrity("transport"), @Authentication, and @Confidentiality("transport"),

715 The effective intent annotation on the helloThere method of the HelloChildService is @Integrity
716 and @Authentication("transport"), the same as in HelloService class.

717 The effective intent annotation on the hello method of the HelloService is @Integrity and
718 @Authentication("message")

719

720 The listing below contains the equivalent declarative security interaction policy of the HelloService
721 and HelloChildService implementation corresponding to the Java interfaces and classes shown in
722 Example 2a.

723

```

724 <?xml version="1.0" encoding="ASCII"?>
725 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
726           name="HelloServiceComposite" >
727     <service name="HelloService" requires="integrity/transport
728           authentication">
729       ...
730     </service>
731     <service name="HelloChildService" requires="integrity/transport
732           authentication confidentiality/message">
733       ...
734     </service>
735     ...
736
737     <component name="HelloServiceComponent">*
738       <implementation.java class="services.hello.HelloService"/>
739       <operation name="hello" requires="integrity
740             authentication/message"/>
741       <operation name="helloThere"
742             requires="integrity
743             authentication/transport"/>
744     </component>
745     <component name="HelloChildServiceComponent">*
746       <implementation.java
747             class="services.hello.HelloChildService" />
748       <operation name="hello"
749             requires="confidentiality/transport"/>
750       <operation name="helloThere" requires=" integrity/transport
751             authentication"/>
752       <operation name="helloWorld" requires="authentication"/>
753     </component>
754

```



```
755     ...
756
757     </composite>
```

758 Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.

760

761 7.4 Relationship of Declarative And Annotated Intents

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

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

769 7.5 Policy Set Annotations

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

773 Policy Sets can be applied directly to Java implementations using the **@PolicySets** annotation.
774 The @PolicySets annotation either takes the QName of a single policy set as a string or the name of
775 two or more policy sets as an array of strings:
776

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

779

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

781 An example of the @PolicySets annotation:

782

```
783     @Reference(name="helloService", required=true)
784     @PolicySets({ MY_NS + "WS_Encryption_Policy",
785                 MY_NS + "WS_Authentication_Policy" })
786     public setHelloService(HelloService service) {
787         ...
788     }
789
```

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

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

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

- 795 • Java class
- 796 • Java interface
- 797 • Method
- 798 • Field
- 799 • Constructor parameter

800 7.6 Security Policy Annotations

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

803 7.6.1 Security Interaction Policy

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

- 806 • @Integrity
- 807 • @Confidentiality
- 808 • @Authentication

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

- 810 • message
- 811 • transport

812 The formal definitions of the @Authentication, @Confidentiality and @Integrity annotations are
813 found in the sections [@Authentication](#), [@Confidentiality](#) and [@Integrity](#).

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

```
817
818 package services.hello;
819 //Interface for HelloService
820 public interface HelloService {
821     String hello(String helloMsg);
822 }
823
824 package services.client;
825 // Interface for ClientService
826 public interface ClientService {
827     public void clientMethod();
828 }
829
830 // Implementation class for ClientService
831 package services.client;
832
833 import services.hello.HelloService;
834 import org.oasisopen.sca.annotation.*;
835
836 @Service(ClientService.class)
837 public class ClientServiceImpl implements ClientService {
838
839     private HelloService helloService;
840
841     @Reference(name="helloService", required=true)
842     @Integrity("message")
843     @Authentication("message")
844     public void setHelloService(HelloService service) {
845         helloService = service;
846     }
847
848     public void clientMethod() {
849         String result = helloService.hello("Hello World!");
```

```
850     ...
851     }
852 }
853
```

854 Example 1. Usage of annotated intents on a reference.

855 7.6.2 Security Implementation Policy

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

- 859 • RunAs
860 Takes as a parameter a string which is the name of a Security role.
861 eg. @RunAs("Manager")
- 863 • Code marked with this annotation will execute with the Security permissions of the
864 identified role.
- 865 • RolesAllowed
866 Takes as a parameter a single string or an array of strings which represent one or more
867 role names. When present, the implementation can only be accessed by principals whose
868 role corresponds to one of the role names listed in the @roles attribute. How role names
869 are mapped to security principals is implementation dependent (SCA does not define this).
870 eg. @RolesAllowed({"Manager", "Employee"})
- 872 • PermitAll
873 No parameters. When present, grants access to all roles.
- 875 • DenyAll
876 No parameters. When present, denies access to all roles.
- 878 • DeclareRoles
879 Takes as a parameter a string or an array of strings which identify one or more role names
880 that form the set of roles used by the implementation.
881 eg. @DeclareRoles({"Manager", "Employee", "Customer"})

882 (all these are declared in the Java package javax.annotation.security)

883 For a full explanation of these intents, see [the Policy Framework specification \[POLICY\]](#).

884 7.6.2.1 Annotated Implementation Policy Example

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

```
886
887 package services.account;
888 @Remotable
889 public interface AccountService {
890     AccountReport getAccountReport(String customerID);
891 }
892
```

893 The following is a full listing of the AccountServiceImpl class, showing the Service it implements,
894 plus the service references it makes and the settable properties that it has, along with a set of
895 implementation policy annotations:

896

```

897 package services.account;
898 import java.util.List;
899 import commonj.sdo.DataFactory;
900 import org.oasisopen.sca.annotation.Property;
901 import org.oasisopen.sca.annotation.Reference;
902 import org.oasisopen.sca.annotation.RolesAllowed;
903 import org.oasisopen.sca.annotation.RunAs;
904 import org.oasisopen.sca.annotation.PermitAll;
905 import services.accountdata.AccountDataService;
906 import services.accountdata.CheckingAccount;
907 import services.accountdata.SavingsAccount;
908 import services.accountdata.StockAccount;
909 import services.stockquote.StockQuoteService;
910 @RolesAllowed("customers")
911 @RunAs("accountants" )
912 public class AccountServiceImpl implements AccountService {
913
914     @Property
915     protected String currency = "USD";
916
917     @Reference
918     protected AccountDataService accountDataService;
919     @Reference
920     protected StockQuoteService stockQuoteService;
921
922     @RolesAllowed({"customers", "accountants"})
923     public AccountReport getAccountReport(String customerID) {
924
925         DataFactory dataFactory = DataFactory.INSTANCE;
926         AccountReport accountReport =
927             (AccountReport)dataFactory.create(AccountReport.class);
928         List accountSummaries = accountReport.getAccountSummaries();
929
930         CheckingAccount checkingAccount =
931             accountDataService.getCheckingAccount(customerID);
932         AccountSummary checkingAccountSummary =
933             (AccountSummary)dataFactory.create(AccountSummary.class);
934         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber()
935 );
936         checkingAccountSummary.setAccountType("checking");
937         checkingAccountSummary.setBalance(fromUSDollarToCurrency
938             (checkingAccount.getBalance()));
939         accountSummaries.add(checkingAccountSummary);
940
941         SavingsAccount savingsAccount =
942             accountDataService.getSavingsAccount(customerID);
943         AccountSummary savingsAccountSummary =
944             (AccountSummary)dataFactory.create(AccountSummary.class);
945         savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
946         savingsAccountSummary.setAccountType("savings");
947         savingsAccountSummary.setBalance(fromUSDollarToCurrency
948             (savingsAccount.getBalance()));
949         accountSummaries.add(savingsAccountSummary);
950
951         StockAccount stockAccount =
952             accountDataService.getStockAccount(customerID);

```

```

955     AccountSummary stockAccountSummary =
956         (AccountSummary)dataFactory.create(AccountSummary.class);
957     stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
958     stockAccountSummary.setAccountType("stock");
959     float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*
960         stockAccount.getQuantity();
961     stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
962     accountSummaries.add(stockAccountSummary);
963
964     return accountReport;
965 }
966
967 @PermitAll
968 public float fromUSDollarToCurrency(float value) {
969
970     if (currency.equals("USD")) return value; else
971     if (currency.equals("EURO")) return value * 0.8f; else
972     return 0.0f;
973 }
974 }

```

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

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

- 977 • @RolesAllowed("customers") - indicating that customers have access to the
- 978 implementation as a whole
- 979 • @RunAs("accountants") – indicating that the code in the implementation runs with the
- 980 permissions of accountants

981 The getAccountReport(..) method is marked with @RolesAllowed({"customers", "accountants"}),
982 which indicates that this method can be called by both customers and accountants.

983 The fromUSDollarToCurrency() method is marked with @PermitAll, which means that this method
984 can be called by any role.

985 8 Java API

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

987 8.1 Component Context

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

```
989
990 package org.oasisopen.sca;
991
992 public interface ComponentContext {
993     String getURI();
994
995     <B> B getService(Class<B> businessInterface, String referenceName);
996
997     <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
998                                             String referenceName);
1000     <B> Collection<B> getServices(Class<B> businessInterface,
1001                                String referenceName);
1002
1003     <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
1004                                                            businessInterface, String referenceName);
1005
1006     <B> ServiceReference<B> createSelfReference(Class<B>
1007                                                businessInterface);
1008
1009     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
1010                                                String serviceName);
1011
1012     <B> B getProperty(Class<B> type, String propertyName);
1013
1014     <B, R extends ServiceReference<B>> R cast(B target)
1015         throws IllegalArgumentException;
1016
1017     RequestContext getRequestContext();
1018
1019
1020 }
```

- 1021
- 1022 • **getURI()** - returns the absolute URI of the component within the SCA domain
 - 1023 • **getService(Class businessInterface, String referenceName)** – Returns a proxy for
1024 the reference defined by the current component. The getService() method takes as its
1025 input arguments the Java type used to represent the target service on the client and the
1026 name of the service reference. It returns an object providing access to the service. The
1027 returned object implements the Java interface the service is typed with. This method
1028 MUST throw an IllegalArgumentException if the reference has multiplicity greater than
1029 one.
 - 1030 • **getServiceReference(Class businessInterface, String referenceName)** – Returns a
1031 ServiceReference defined by the current component. This method MUST throw an
1032 IllegalArgumentException if the reference has multiplicity greater than one.

- 1033 • **getServices(Class businessInterface, String referenceName)** – Returns a list of
1034 typed service proxies for a business interface type and a reference name.
- 1035 • **getServiceReferences(Class businessInterface, String referenceName)** –Returns a
1036 list typed service references for a business interface type and a reference name.
- 1037 • **createSelfReference(Class businessInterface)** – Returns a ServiceReference that can
1038 be used to invoke this component over the designated service.
- 1039 • **createSelfReference(Class businessInterface, String serviceName)** – Returns a
1040 ServiceReference that can be used to invoke this component over the designated service.
1041 Service name explicitly declares the service name to invoke
- 1042 • **getProperty (Class type, String propertyName)** - Returns the value of an SCA
1043 property defined by this component.
- 1044 • **getRequestContext()** - Returns the context for the current SCA service request, or null if
1045 there is no current request or if the context is unavailable. This method MUST return non-
1046 null when invoked during the execution of a Java business method for a service operation
1047 or callback operation, on the same thread that the SCA runtime provided, and MUST
1048 return null in all other cases.
- 1049 • **cast(B target)** - Casts a type-safe reference to a ServiceReference

1050 A component can access its component context by defining a field or setter method typed by
1051 **org.oasisopen.sca.ComponentContext** and annotated with **@Context**. To access the target
1052 service, the component uses **ComponentContext.getService(..)**.

1053 The following shows an example of component context usage in a Java class using the @Context
1054 annotation.

```
1055 private ComponentContext componentContext;
1056
1057 @Context
1058 public void setContext(ComponentContext context) {
1059     componentContext = context;
1060 }
1061
1062 public void doSomething() {
1063     HelloWorld service =
1064     componentContext.getService(HelloWorld.class, "HelloWorldComponent");
1065     service.hello("hello");
1066 }
1067
```

1068 Similarly, non-SCA client code can use the ComponentContext API to perform operations against a
1069 component in an SCA domain. How the non-SCA client code obtains a reference to a
1070 ComponentContext is runtime specific.

1071 8.2 Request Context

1072 The following shows the **RequestContext** interface:

```
1073
1074 package org.oasisopen.sca;
1075
1076 import javax.security.auth.Subject;
1077
1078 public interface RequestContext {
1079
1080     Subject getSecuritySubject();
1081
1082     String getServiceName();

```

```

1083     <CB> ServiceReference<CB> getCallbackReference();
1084     <CB> CB getCallback();
1085     <B> ServiceReference<B> getServiceReference();
1086
1087 }
1088

```

1089 The RequestContext interface has the following methods:

- 1090 • **getSecuritySubject()** – Returns the JAAS Subject of the current request
- 1091 • **getServiceName()** – Returns the name of the service on the Java implementation the
1092 request came in on
- 1093 • **getCallbackReference()** – Returns a service reference to the callback as specified by the
1094 caller. This method returns null when called for a service request whose interface is not
1095 bidirectional or when called for a callback request.
- 1096 • **getCallback()** – Returns a proxy for the callback as specified by the caller. Similar to the
1097 getCallbackReference() method, this method returns null when called for a service request
1098 whose interface is not bidirectional or when called for a callback request.
- 1099 • **getServiceReference()** – When invoked during the execution of a service operation, this
1100 method MUST return a ServiceReference that represents the service that was invoked.
1101 When invoked during the execution of a callback operation, this method MUST return a
1102 CallableReference that represents the callback that was invoked.

1103 8.3 ServiceReference

1104 ServiceReferences can be injected using the @Reference annotation on a field, a setter method,
1105 or constructor parameter taking the type ServiceReference. The detailed description of the usage
1106 of these methods is described in the section on Asynchronous Programming in this document.

1107 The following Java code defines the **ServiceReference** interface:

```

1108 package org.oasisopen.sca;
1109
1110 public interface ServiceReference<B> extends java.io.Serializable {
1111
1112     B getService();
1113     Class<B> getBusinessInterface();
1114 }
1115

```

1116 The ServiceReference interface has the following methods:

- 1117
- 1118 • **getService()** - Returns a type-safe reference to the target of this reference. The instance
1119 returned is guaranteed to implement the business interface for this reference. The value
1120 returned is a proxy to the target that implements the business interface associated with this
1121 reference.
- 1122 • **getBusinessInterface()** – Returns the Java class for the business interface associated with
1123 this reference.

1124 8.4 ServiceRuntimeException

1125 The following snippet shows the **ServiceRuntimeException**.

```

1126
1127 package org.oasisopen.sca;
1128
1129 public class ServiceRuntimeException extends RuntimeException {

```


1130 ...
1131 }

1132 This exception signals problems in the management of SCA component execution.
1133

1134 **8.5 ServiceUnavailableException**

1135 The following snippet shows the *ServiceUnavailableException*.

```
1136        package org.oasisopen.sca;  
1137  
1138        public class ServiceUnavailableException extends ServiceRuntimeException {  
1139            ...  
1140        }  
1141  
1142
```

1143 This exception signals problems in the interaction with remote services. These are exceptions
1144 that can be transient, so retrying is appropriate. Any exception that is a *ServiceRuntimeException*
1145 that is *not* a *ServiceUnavailableException* is unlikely to be resolved by retrying the operation, since
1146 it most likely requires human intervention

1147 **8.6 InvalidServiceException**

1148 The following snippet shows the *InvalidServiceException*.

```
1149        package org.oasisopen.sca;  
1150  
1151        public class InvalidServiceException extends ServiceRuntimeException {  
1152            ...  
1153        }  
1154  
1155
```

1156 This exception signals that the *ServiceReference* is no longer valid. This can happen when the
1157 target of the reference is undeployed. This exception is not transient and therefore is unlikely to
1158 be resolved by retrying the operation and will most likely require human intervention.

1159 **8.7 Constants**

1160 The SCA *Constants* interface defines a number of constant values that are used in the SCA Java
1161 APIs and Annotations. The following snippet shows the *Constants* interface:

```
1162        package org.oasisopen.sca;  
1163  
1164        public interface Constants {  
1165            String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";  
1166            String SCA_PREFIX = "{"+SCA_NS+"}";  
1167        }  
1168
```

1169 9 Java Annotations

1170 This section provides definitions of all the Java annotations which apply to SCA.

1171 This specification places constraints on some annotations that are not detectable by a Java
1172 compiler. For example, the definition of the `@Property` and `@Reference` annotations indicate that
1173 they are allowed on parameters, but sections 8.14 and 8.15 constrain those definitions to
1174 constructor parameters. An SCA runtime MUST verify the proper use of all annotations and if an
1175 annotation is improperly used, the SCA runtime MUST NOT run the component which uses the
1176 invalid implementation code.

1177 SCA annotations are not allowed on static methods and static fields. It is an error to use an SCA
1178 annotation on a static method or a static field of an implementation class and the SCA runtime
1179 MUST NOT instantiate such an implementation class.

1180 9.1 @AllowsPassByReference

1181 The following Java code defines the `@AllowsPassByReference` annotation:

1182

```
1183 package org.oasisopen.sca.annotation;  
1184  
1185 import static java.lang.annotation.ElementType.FIELD;  
1186 import static java.lang.annotation.ElementType.METHOD;  
1187 import static java.lang.annotation.ElementType.PARAMETER;  
1188 import static java.lang.annotation.ElementType.TYPE;  
1189 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1190 import java.lang.annotation.Retention;  
1191 import java.lang.annotation.Target;  
1192  
1193 @Target({TYPE, METHOD, FIELD, PARAMETER})  
1194 @Retention(RUNTIME)  
1195 public @interface AllowsPassByReference {  
1196  
1197     boolean value() default true;  
1198 }  
1199
```

1200 The `@AllowsPassByReference` annotation allows service method implementations and client
1201 references to be marked as “allows pass by reference” to indicate that they use input parameters,
1202 return values and exceptions in a manner that allows the SCA runtime to avoid the cost of copying
1203 mutable objects when a remotable service is called locally within the same JVM.

1204 The `@AllowsPassByReference` annotation has the following attribute:

- 1205 • value – specifies whether the “allows pass by reference” marker applies to the service
1206 implementation class, service implementation method, or client reference to which this
1207 annotation applies; if not specified, defaults to true.

1208 The `@AllowsPassByReference` annotation MAY be placed on an individual method of a remotable
1209 service implementation, on a service implementation class, or on an individual reference for a
1210 remotable service. When applied to a reference, it MAY appear anywhere that the `@Remotable`
1211 annotation MAY appear. It MUST NOT appear anywhere else.

1212 The “allows pass by reference” marking of a method implementation of a remotable service is
1213 determined as follows:

- 1214 1. If the method has an `@AllowsPassByReference` annotation, the method is marked “allows
1215 pass by reference” if and only if the value of the method’s annotation is true.

Deleted: TYPE

Deleted: The `@AllowsPassByReference` annotation is used on implementations of remotable interfaces to indicate that interactions with the service from a client within the same address space are allowed to use pass by reference data exchange semantics. The implementation promises that its by-value semantics will be maintained even if the parameters and return values are actually passed by-reference. This means that the service will not modify any operation input parameter or return value, even after returning from the operation. Either a whole class implementing a remotable service or an individual remotable service method implementation can be annotated using the `@AllowsPassByReference` annotation.¶
`@AllowsPassByReference` has no attributes

Formatted: Font: Not Bold, Not Italic

Formatted: Bulleted + Level: 1 +
Aligned at: 36 pt + Tab after: 54 pt
+ Indent at: 54 pt

1244 2. Otherwise, if the class has an @AllowsPassByReference annotation, the method is marked
1245 "allows pass by reference" if and only if the value of the class's annotation is true.

1246 3. Otherwise, the method is not marked "allows pass by reference".

1247 The "allows pass by reference" marking of a reference for a remotable service is determined as
1248 follows:

1249 1. If the reference has an @AllowsPassByReference annotation, the reference is marked
1250 "allows pass by reference" if and only if the value of the reference's annotation is true.

1251 2. Otherwise, if the service implementation class containing the reference has an
1252 @AllowsPassByReference annotation, the reference is marked "allows pass by reference" if
1253 and only if the value of the class's annotation is true.

1254 3. Otherwise, the reference is not marked "allows pass by reference".

1255 ▼ ----- Deleted: ¶

1256 The following snippet shows a sample where @AllowsPassByReference is defined for the
1257 implementation of a service method on the Java component implementation class.

1258

```
1259 @AllowsPassByReference  
1260 public String hello(String message) {  
1261     ...  
1262 }  
1263
```

1264 The following snippet shows a sample where @AllowsPassByReference is defined for a client
1265 reference of a Java component implementation class.

```
1266 @AllowsPassByReference  
1267 @Reference  
1268 private StockQuoteService stockQuote;
```

Formatted: Normal, Indent: First
line: 18 pt, Don't adjust space
between Latin and Asian text, Don't
adjust space between Asian text and

1269 9.2 @Authentication

1270 The following Java code defines the **@Authentication** annotation:

```
1271 package org.oasisopen.sca.annotation;  
1272  
1273 import static java.lang.annotation.ElementType.FIELD;  
1274 import static java.lang.annotation.ElementType.METHOD;  
1275 import static java.lang.annotation.ElementType.PARAMETER;  
1276 import static java.lang.annotation.ElementType.TYPE;  
1277 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1278 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
1279  
1280 import java.lang.annotation.Inherited;  
1281 import java.lang.annotation.Retention;  
1282 import java.lang.annotation.Target;  
1283  
1284 @Inherited  
1285 @Target({TYPE, FIELD, METHOD, PARAMETER})  
1286 @Retention(RUNTIME)  
1287 @Intent(Authentication.AUTHENTICATION)  
1288 public @interface Authentication {  
1289     String AUTHENTICATION = SCA_PREFIX + "authentication";  
1290     String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";  
1291     String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";  
1292  
1293
```

```

1295     /**
1296     * List of authentication qualifiers (such as "message"
1297     * or "transport").
1298     *
1299     * @return authentication qualifiers
1300     */
1301     @Qualifier
1302     String[] value() default "";
1303 }

```

1304 The **@Authentication** annotation is used to indicate that the invocation requires authentication.
1305 See the [section on Application of Intent Annotations](#) for samples and details.

1306 9.3 @Callback

1307 The following Java code defines shows the **@Callback** annotation:

```

1308
1309 package org.oasisopen.sca.annotation;
1310
1311 import static java.lang.annotation.ElementType.TYPE;
1312 import static java.lang.annotation.ElementType.METHOD;
1313 import static java.lang.annotation.ElementType.FIELD;
1314 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1315 import java.lang.annotation.Retention;
1316 import java.lang.annotation.Target;
1317
1318 @Target(TYPE, METHOD, FIELD)
1319 @Retention(RUNTIME)
1320 public @interface Callback {
1321
1322     Class<?> value() default Void.class;
1323 }
1324
1325

```

1326 The @Callback annotation is used to annotate a service interface with a callback interface, which
1327 takes the Java Class object of the callback interface as a parameter.

1328 The @Callback annotation has the following attribute:

- 1329 • **value** – the name of a Java class file containing the callback interface

1330

1331 The @Callback annotation can also be used to annotate a method or a field of an SCA
1332 implementation class, in order to have a callback object injected

1333

1334 The following snippet shows a @Callback annotation on an interface:

1335

```

1336 @Remotable
1337 @Callback(MyServiceCallback.class)
1338 public interface MyService {
1339
1340     void someAsyncMethod(String arg);
1341 }
1342

```

1343 An example use of the @Callback annotation to declare a callback interface follows:

1344

```

1345 package somepackage;
1346 import org.oasisopen.sca.annotation.Callback;
1347 import org.oasisopen.sca.annotation.Remotable;
1348 @Remotable
1349 @Callback(MyServiceCallback.class)
1350 public interface MyService {
1351
1352     void someMethod(String arg);
1353 }
1354
1355 @Remotable
1356 public interface MyServiceCallback {
1357
1358     void receiveResult(String result);
1359 }

```

1360 In this example, the implied component type is:

```

1361
1362
1363 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >
1364
1365     <service name="MyService">
1366         <interface.java interface="somepackage.MyService"
1367             callbackInterface="somepackage.MyServiceCallback" />
1368     </service>
1369 </componentType>

```

1370 9.4 @ComponentName

1371 The following Java code defines the **@ComponentName** annotation:

```

1372
1373 package org.oasisopen.sca.annotation;
1374
1375 import static java.lang.annotation.ElementType.METHOD;
1376 import static java.lang.annotation.ElementType.FIELD;
1377 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1378 import java.lang.annotation.Retention;
1379 import java.lang.annotation.Target;
1380
1381 @Target({METHOD, FIELD})
1382 @Retention(RUNTIME)
1383 public @interface ComponentName {
1384
1385 }
1386

```

1387 The @ComponentName annotation is used to denote a Java class field or setter method that is
1388 used to inject the component name.

1389 The following snippet shows a component name field definition sample.

```

1390
1391 @ComponentName
1392 private String componentName;
1393

```

1394 The following snippet shows a component name setter method sample.

```
1395
1396 @ComponentName
1397 public void setComponentName(String name) {
1398     //...
1399 }
```

1400 9.5 @Confidentiality

1401 The following Java code defines the *@Confidentiality* annotation:

```
1402 package org.oasisopen.sca.annotations;
1403
1404 import static java.lang.annotation.ElementType.FIELD;
1405 import static java.lang.annotation.ElementType.METHOD;
1406 import static java.lang.annotation.ElementType.PARAMETER;
1407 import static java.lang.annotation.ElementType.TYPE;
1408 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1409 import static org.oasisopen.sca.Constants.SCA_PREFIX;
1410
1411 import java.lang.annotation.Inherited;
1412 import java.lang.annotation.Retention;
1413 import java.lang.annotation.Target;
1414
1415 @Inherited
1416 @Target({TYPE, FIELD, METHOD, PARAMETER})
1417 @Retention(RUNTIME)
1418 @Intent(Confidentiality.CONFIDENTIALITY)
1419 public @interface Confidentiality {
1420     String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
1421     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
1422     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";
1423
1424     /**
1425      * List of confidentiality qualifiers (such as "message" or
1426      * "transport").
1427      *
1428      * @return confidentiality qualifiers
1429      */
1430     @Qualifier
1431     String[] value() default "";
1432 }
1433
```

1434 The *@Confidentiality* annotation is used to indicate that the invocation requires confidentiality.

1435 See the [section on Application of Intent Annotations](#) for samples and details.

1436 9.6 @Constructor

1437 The following Java code defines the *@Constructor* annotation:

```
1438 package org.oasisopen.sca.annotation;
1439
1440 import static java.lang.annotation.ElementType.CONSTRUCTOR;
1441 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1442 import java.lang.annotation.Retention;
1443 import java.lang.annotation.Target;
1444
1445 @Target(CONSTRUCTOR)
```

```
1447 @Retention(RUNTIME)
1448 public @interface Constructor { }
1449
```

1450 The @Constructor annotation is used to mark a particular constructor to use when instantiating a
1451 Java component implementation. If this constructor has parameters, each of these parameters
1452 MUST have either a @Property annotation or a @Reference annotation.

1453 The following snippet shows a sample for the @Constructor annotation.

1454

```
1455 public class HelloServiceImpl implements HelloService {
1456
1457     public HelloServiceImpl(){
1458         ...
1459     }
1460
1461     @Constructor
1462     public HelloServiceImpl(@Property(name="someProperty") String
1463     someProperty ){
1464         ...
1465     }
1466
1467     public String hello(String message) {
1468         ...
1469     }
1470 }
```

1471 9.7 @Context

1472 The following Java code defines the **@Context** annotation:

1473

```
1474 package org.oasisopen.sca.annotation;
1475
1476 import static java.lang.annotation.ElementType.METHOD;
1477 import static java.lang.annotation.ElementType.FIELD;
1478 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1479 import java.lang.annotation.Retention;
1480 import java.lang.annotation.Target;
1481
1482 @Target({METHOD, FIELD})
1483 @Retention(RUNTIME)
1484 public @interface Context {
1485
1486 }
1487
```

1488 The @Context annotation is used to denote a Java class field or a setter method that is used to
1489 inject a composite context for the component. The type of context to be injected is defined by the
1490 type of the Java class field or type of the setter method input argument; the type is either
1491 **ComponentContext** or **RequestContext**.

1492 The @Context annotation has no attributes.

1493

1494 The following snippet shows a ComponentContext field definition sample.

1495

```
1496 @Context
```

```
1497     protected ComponentContext context;
1498
```

1499 The following snippet shows a RequestContext field definition sample.

```
1500
1501     @Context
1502     protected RequestContext context;
```

1503 9.8 @Destroy

1504 The following Java code defines the **@Destroy** annotation:

```
1505
1506     package org.oasisopen.sca.annotation;
1507
1508     import static java.lang.annotation.ElementType.METHOD;
1509     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1510     import java.lang.annotation.Retention;
1511     import java.lang.annotation.Target;
1512
1513     @Target(METHOD)
1514     @Retention(RUNTIME)
1515     public @interface Destroy {
1516
1517     }
1518
```

1519 The @Destroy annotation is used to denote a single Java class method that will be called when the
1520 scope defined for the implementation class ends. The method MAY have any access modifier and
1521 MUST have a void return type and no arguments.

1522 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1523 when the scope defined for the implementation class ends. If the implementation class has a
1524 method with an @Destroy annotation that does not match these criteria, the SCA runtime MUST
1525 NOT instantiate the implementation class.

1526 The following snippet shows a sample for a destroy method definition.

```
1527
1528     @Destroy
1529     public void myDestroyMethod() {
1530         ...
1531     }
```

1532 9.9 @EagerInit

1533 The following Java code defines the **@EagerInit** annotation:

```
1534
1535     package org.oasisopen.sca.annotation;
1536
1537     import static java.lang.annotation.ElementType.TYPE;
1538     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1539     import java.lang.annotation.Retention;
1540     import java.lang.annotation.Target;
1541
1542     @Target(TYPE)
1543     @Retention(RUNTIME)
1544     public @interface EagerInit {
```


1545
1546 }
1547
1548 The *@EagerInit* annotation is used to annotate the Java class of a COMPOSITE scoped
1549 implementation for eager initialization. When marked for eager initialization, the composite scoped
1550 instance is created when its containing component is started.

1551 9.10 @Init

1552 The following Java code defines the *@Init* annotation:

```
1553 package org.oasisopen.sca.annotation;  
1554  
1555 import static java.lang.annotation.ElementType.METHOD;  
1556 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1557 import java.lang.annotation.Retention;  
1558 import java.lang.annotation.Target;  
1559  
1560 @Target(METHOD)  
1561 @Retention(RUNTIME)  
1562 public @interface Init {  
1563  
1564 }  
1565  
1566 }  
1567
```

1568 The *@Init* annotation is used to denote a single Java class method that is called when the scope
1569 defined for the implementation class starts. The method MAY have any access modifier and MUST
1570 have a void return type and no arguments.

1571 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1572 after all property and reference injection is complete. If the implementation class has a method
1573 with an *@Init* annotation that does not match these criteria, the SCA runtime MUST NOT
1574 instantiate the implementation class.

1575 The following snippet shows an example of an init method definition.

```
1576  
1577 @Init  
1578 public void myInitMethod() {  
1579     ...  
1580 }
```

1581 9.11 @Integrity

1582 The following Java code defines the *@Integrity* annotation:

```
1583 package org.oasisopen.sca.annotation;  
1584  
1585 import static java.lang.annotation.ElementType.FIELD;  
1586 import static java.lang.annotation.ElementType.METHOD;  
1587 import static java.lang.annotation.ElementType.PARAMETER;  
1588 import static java.lang.annotation.ElementType.TYPE;  
1589 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1590 import static org.oasisopen.Constants.SCA_PREFIX;  
1591  
1592 import java.lang.annotation.Inherited;  
1593 import java.lang.annotation.Retention;  
1594
```

```

1595 import java.lang.annotation.Target;
1596
1597 @Inherited
1598 @Target({TYPE, FIELD, METHOD, PARAMETER})
1599 @Retention(RUNTIME)
1600 @Intent(Integrity.INTEGRITY)
1601 public @interface Integrity {
1602     String INTEGRITY = SCA_PREFIX + "integrity";
1603     String INTEGRITY_MESSAGE = INTEGRITY + ".message";
1604     String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";
1605
1606     /**
1607      * List of integrity qualifiers (such as "message" or "transport").
1608      *
1609      * @return integrity qualifiers
1610      */
1611     @Qualifier
1612     String[] value() default "";
1613 }
1614

```

1615 The **@Integrity** annotation is used to indicate that the invocation requires integrity (ie no
1616 tampering of the messages between client and service).

1617 See the [section on Application of Intent Annotations](#) for samples and details.

1618 9.12 @Intent

1619 The following Java code defines the **@Intent** annotation:

```

1620 package org.oasisopen.sca.annotation;
1621
1622 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
1623 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1624 import java.lang.annotation.Retention;
1625 import java.lang.annotation.Target;
1626
1627 @Target({ANNOTATION_TYPE})
1628 @Retention(RUNTIME)
1629 public @interface Intent {
1630     /**
1631      * The qualified name of the intent, in the form defined by
1632      * {@link javax.xml.namespace.QName#toString}.
1633      * @return the qualified name of the intent
1634      */
1635     String value() default "";
1636
1637     /**
1638      * The XML namespace for the intent.
1639      * @return the XML namespace for the intent
1640      */
1641     String targetNamespace() default "";
1642
1643     /**
1644      * The name of the intent within its namespace.
1645      * @return name of the intent within its namespace
1646      */
1647     String localPart() default "";
1648 }
1649

```

1650

1651 The `@Intent` annotation is used for the creation of new annotations for specific intents. It is not
1652 expected that the `@Intent` annotation will be used in application code.

1653 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to
1654 define new intent annotations.

1655 9.13 @OneWay

1656 The following Java code defines the `@OneWay` annotation:

1657

```
1658 package org.oasisopen.sca.annotation;  
1659  
1660 import static java.lang.annotation.ElementType.METHOD;  
1661 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1662 import java.lang.annotation.Retention;  
1663 import java.lang.annotation.Target;  
1664  
1665 @Target(METHOD)  
1666 @Retention(RUNTIME)  
1667 public @interface OneWay {  
1668  
1669  
1670 }  
1671
```

1672 The `@OneWay` annotation is used on a Java interface or class method to indicate that invocations
1673 will be dispatched in a non-blocking fashion as described in the section on Asynchronous
1674 Programming.

1675 The `@OneWay` annotation has no attributes.

1676 The following snippet shows the use of the `@OneWay` annotation on an interface.

```
1677 package services.hello;  
1678  
1679 import org.oasisopen.sca.annotation.OneWay;  
1680  
1681 public interface HelloService {  
1682     @OneWay  
1683     void hello(String name);  
1684 }
```

1685 9.14 @PolicySet

1686 The following Java code defines the `@PolicySets` annotation:

1687

```
1688 package org.oasisopen.sca.annotation;  
1689  
1690 import static java.lang.annotation.ElementType.FIELD;  
1691 import static java.lang.annotation.ElementType.METHOD;  
1692 import static java.lang.annotation.ElementType.PARAMETER;  
1693 import static java.lang.annotation.ElementType.TYPE;  
1694 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1695  
1696 import java.lang.annotation.Retention;  
1697 import java.lang.annotation.Target;  
1698  
1699 @Target({TYPE, FIELD, METHOD, PARAMETER})
```

```

1700 @Retention(RUNTIME)
1701 public @interface PolicySets {
1702     /**
1703      * Returns the policy sets to be applied.
1704      *
1705      * @return the policy sets to be applied
1706      */
1707     String[] value() default "";
1708 }
1709

```

1710 The `@PolicySet` annotation is used to attach an SCA Policy Set to a Java implementation class or
1711 to one of its subelements.

1712 See the [section "Policy Set Annotations"](#) for details and samples.

1713 9.15 @Property

1714 The following Java code defines the `@Property` annotation:

```

1715 package org.oasisopen.sca.annotation;
1716
1717 import static java.lang.annotation.ElementType.METHOD;
1718 import static java.lang.annotation.ElementType.FIELD;
1719 import static java.lang.annotation.ElementType.PARAMETER;
1720 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1721 import java.lang.annotation.Retention;
1722 import java.lang.annotation.Target;
1723
1724 @Target({METHOD, FIELD, PARAMETER})
1725 @Retention(RUNTIME)
1726 public @interface Property {
1727
1728     String name() default "";
1729     boolean required() default true;
1730 }
1731

```

1732 The `@Property` annotation is used to denote a Java class field, a setter method, or a constructor
1733 parameter that is used to inject an SCA property value. The type of the property injected, which
1734 can be a simple Java type or a complex Java type, is defined by the type of the Java class field or
1735 the type of the input parameter of the setter method or constructor.

1736 The `@Property` annotation can be used on fields, on setter methods or on a constructor method
1737 parameter. However, the `@Property` annotation MUST NOT be used on a class field that is declared
1738 as final.

1739 Properties can also be injected via setter methods even when the `@Property` annotation is not
1740 present. However, the `@Property` annotation must be used in order to inject a property onto a
1741 non-public field. In the case where there is no `@Property` annotation, the name of the property is
1742 the same as the name of the field or setter.

1743 Where there is both a setter method and a field for a property, the setter method is used.

1744 The `@Property` annotation has the following attributes:

- 1745 • **name (optional)** – the name of the property. For a field annotation, the default is the
1746 name of the field of the Java class. For a setter method annotation, the default is the
1747 JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a
1748 constructor parameter annotation, there is no default and the name attribute MUST be
1749 present.
- 1750 • **required (optional)** – specifies whether injection is required, defaults to true. For a
1751 constructor parameter annotation, this attribute MUST have the value true.

1752
1753 The following snippet shows a property field definition sample.

```
1754  
1755 @Property(name="currency", required=true)  
1756 protected String currency;
```

1757
1758 The following snippet shows a property setter sample

```
1759  
1760 @Property(name="currency", required=true)  
1761 public void setCurrency( String theCurrency ) {  
1762     ....  
1763 }
```

1764
1765 If the property is defined as an array or as any type that extends or implements
1766 **java.util.Collection**, then the implied component type has a property with a **many** attribute set to
1767 true.

1768 The following snippet shows the definition of a configuration property using the @Property
1769 annotation for a collection.

```
1770 ...  
1771 private List<String> helloConfigurationProperty;  
1772  
1773 @Property(required=true)  
1774 public void setHelloConfigurationProperty(List<String> property) {  
1775     helloConfigurationProperty = property;  
1776 }  
1777 ...
```

1778 9.16 @Qualifier

1779 The following Java code defines the **@Qualifier** annotation:

```
1780 package org.oasisopen.sca.annotation;  
1781  
1782 import static java.lang.annotation.ElementType.METHOD;  
1783 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1784  
1785 import java.lang.annotation.Retention;  
1786 import java.lang.annotation.Target;  
1787  
1788 @Target(METHOD)  
1789 @Retention(RUNTIME)  
1790 public @interface Qualifier {  
1791 }  
1792  
1793
```

1794 The @Qualifier annotation is applied to an attribute of a specific intent annotation definition,
1795 defined using the @Intent annotation, to indicate that the attribute provides qualifiers for the
1796 intent. The @Qualifier annotation MUST be used in a specific intent annotation definition where the
1797 intent has qualifiers.

1798 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to
1799 define new intent annotations.

1800 9.17 @Reference

1801 The following Java code defines the **@Reference** annotation:

1802

```
1803 package org.oasisopen.sca.annotation;
1804
1805 import static java.lang.annotation.ElementType.METHOD;
1806 import static java.lang.annotation.ElementType.FIELD;
1807 import static java.lang.annotation.ElementType.PARAMETER;
1808 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1809 import java.lang.annotation.Retention;
1810 import java.lang.annotation.Target;
1811 @Target({METHOD, FIELD, PARAMETER})
1812 @Retention(RUNTIME)
1813 public @interface Reference {
1814
1815     String name() default "";
1816     boolean required() default true;
1817 }
1818
```

1819 The @Reference annotation type is used to annotate a Java class field, a setter method, or a
1820 constructor parameter that is used to inject a service that resolves the reference. The interface of
1821 the service injected is defined by the type of the Java class field or the type of the input parameter
1822 of the setter method or constructor.

1823 The @Reference annotation MUST NOT be used on a class field that is declared as final.

1824 References can also be injected via setter methods even when the @Reference annotation is not
1825 present. However, the @Reference annotation must be used in order to inject a reference onto a
1826 non-public field. In the case where there is no @Reference annotation, the name of the reference
1827 is the same as the name of the field or setter.

1828 Where there is both a setter method and a field for a reference, the setter method is used.

1829 The @Reference annotation has the following attributes:

- 1830 • **name (optional)** – the name of the reference. For a field annotation, the default is the
1831 name of the field of the Java class. For a setter method annotation, the default is the
1832 JavaBeans property name corresponding to the setter method name. For a constructor
1833 parameter annotation, there is no default and the name attribute MUST be present.
- 1834 • **required (optional)** – whether injection of service or services is required. Defaults to true.
1835 For a constructor parameter annotation, this attribute MUST have the value true.

1836

1837 The following snippet shows a reference field definition sample.

1838

```
1839 @Reference(name="stockQuote", required=true)
1840 protected StockQuoteService stockQuote;
```

1841

1842 The following snippet shows a reference setter sample

1843

```
1844 @Reference(name="stockQuote", required=true)
1845 public void setStockQuote( StockQuoteService theSQService ) {
1846     ...
1847 }
```

1848

1849 The following fragment from a component implementation shows a sample of a service reference
1850 using the @Reference annotation. The name of the reference is "helloService" and its type is
1851 HelloService. The clientMethod() calls the "hello" operation of the service referenced by the
1852 helloService reference.

1853

```
1854 package services.hello;
1855
1856 private HelloService helloService;
1857
1858 @Reference(name="helloService", required=true)
1859 public setHelloService(HelloService service) {
1860     helloService = service;
1861 }
1862
1863 public void clientMethod() {
1864     String result = helloService.hello("Hello World!");
1865     ...
1866 }
1867
```

1868 The presence of a @Reference annotation is reflected in the componentType information that the
1869 runtime generates through reflection on the implementation class. The following snippet shows
1870 the component type for the above component implementation fragment.

1871

```
1872 <?xml version="1.0" encoding="ASCII"?>
1873 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1874
1875     <!-- Any services offered by the component would be listed here -->
1876     <reference name="helloService" multiplicity="1..1">
1877         <interface.java interface="services.hello.HelloService"/>
1878     </reference>
1879
1880 </componentType>
1881
```

1882 If the reference is not an array or collection, then the implied component type has a reference
1883 with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference **required**
1884 attribute – 1..1 applies if required=true.

1885

1886 If the reference is defined as an array or as any type that extends or implements **java.util.Collection**,
1887 then the implied component type has a reference with a **multiplicity** of either **1..n** or **0..n**, depending
1888 on whether the **required** attribute of the @Reference annotation is set to true or false – 1..n applies if
1889 required=true.

1890

1891 The following fragment from a component implementation shows a sample of a service reference
1892 definition using the @Reference annotation on a java.util.List. The name of the reference is
1893 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
1894 services referenced by the helloServices reference. In this case, at least one HelloService should
1895 be present, so **required** is true.

1896

```
1897 @Reference(name="helloServices", required=true)
1898 protected List<HelloService> helloServices;
1899
1900 public void clientMethod() {
```

```

1901
1902     ...
1903     for (int index = 0; index < helloServices.size(); index++) {
1904         HelloService helloService =
1905             (HelloService)helloServices.get(index);
1906         String result = helloService.hello("Hello World!");
1907     }
1908     ...
1909 }
1910

```

1911 The following snippet shows the XML representation of the component type reflected from for the
1912 former component implementation fragment. There is no need to author this component type in
1913 this case since it can be reflected from the Java class.

```

1914
1915 <?xml version="1.0" encoding="ASCII"?>
1916 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1917
1918     <!-- Any services offered by the component would be listed here -->
1919     <reference name="helloServices" multiplicity="1..n">
1920         <interface.java interface="services.hello.HelloService"/>
1921     </reference>
1922
1923 </componentType>
1924

```

1925 At runtime, the representation of an unwired reference depends on the reference's multiplicity. An
1926 unwired reference with a multiplicity of 0..1 must be null. An unwired reference with a multiplicity
1927 of 0..N must be an empty array or collection.

1928 9.17.1 Reinjection

1929 References MAY be reinjected after the initial creation of a component if the reference target
1930 changes due to a change in wiring that has occurred since the component was initialized. In order
1931 for reinjection to occur, the following MUST be true:

- 1932 1. The component MUST NOT be STATELESS scoped.
- 1933 2. The reference MUST use either field-based injection or setter injection. References that are
1934 injected through constructor injection MUST NOT be changed. Setter injection allows for
1935 code in the setter method to perform processing in reaction to a change.

1936 If a reference target changes and the reference is not reinjected, the reference MUST continue to
1937 work as if the reference target was not changed.

1938 If an operation is called on a reference where the target of that reference has been undeployed,
1939 the SCA runtime SHOULD throw `InvalidServiceException`. If an operation is called on a reference
1940 where the target of the reference has become unavailable for some reason, the SCA runtime
1941 SHOULD throw `ServiceUnavailableException`. If the target of the reference is changed, the
1942 reference MAY continue to work, depending on the runtime and the type of change that was made.
1943 If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.

1944 A `ServiceReference` that has been obtained from a reference by `ComponentContext.cast()`
1945 corresponds to the reference that is passed as a parameter to `cast()`. If the reference is
1946 subsequently reinjected, the `ServiceReference` obtained from the original reference MUST continue
1947 to work as if the reference target was not changed. If the target of a `ServiceReference` has been
1948 undeployed, the SCA runtime SHOULD throw `InvalidServiceException` when an operation is
1949 invoked on the `ServiceReference`. If the target of a `ServiceReference` has become unavailable, the
1950 SCA runtime SHOULD throw `ServiceUnavailableException` when an operation is invoked on the
1951 `ServiceReference`. If the target of a `ServiceReference` is changed, the reference MAY continue to
1952 work, depending on the runtime and the type of change that was made. If it doesn't work, the
1953 exception thrown will depend on the runtime and the cause of the failure.

1954 A reference or ServiceReference accessed through the component context by calling getService()
 1955 or getServiceReference() MUST correspond to the current configuration of the domain. This
 1956 applies whether or not reinjection has taken place. If the target has been undeployed or has
 1957 become unavailable, the result SHOULD be a reference to the undeployed or unavailable service,
 1958 and attempts to call business methods SHOULD throw an exception as described above. If the
 1959 target has changed, the result SHOULD be a reference to the changed service.

1960 The rules for reference reinjection also apply to references with a multiplicity of 0..N or 1..N. This
 1961 means that in the cases listed above where reference reinjection is not allowed, the array or
 1962 Collection for the reference MUST NOT change its contents. In cases where the contents of a
 1963 reference collection MAY change, then for references that use setter injection, the setter method
 1964 MUST be called for any change to the contents. The reinjected array or Collection MUST NOT be
 1965 the same array or Collection object previously injected to the component.

1966

	Effect on		
Change event	Reference	Existing ServiceReference Object	Subsequent invocations of ComponentContext.getServiceReference() or getService()
Change to the target of the reference	MAY be reinjected (if other conditions* apply). If not reinjected, then it MUST continue to work as if the reference target was not changed.	MUST continue to work as if the reference target was not changed.	Result corresponds to the current configuration of the domain.
Target service undeployed	Business methods SHOULD throw InvalidServiceException.	Business methods SHOULD throw InvalidServiceException.	Result SHOULD be a reference to the undeployed or unavailable service. Business methods SHOULD throw InvalidServiceException.
Target service changed	MAY continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.	MAY continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.	Result SHOULD be a reference to the changed service.
* Other conditions: <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. ** Result of invoking ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().			

1967

1968 9.18 @Remotable

1969 The following Java code defines the **@Remotable** annotation:

1970

```
1971 package org.oasisopen.sca.annotation;
```

1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024

```
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Remotable {
```

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

The @Remotable annotation has no attributes.

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

```
package services.hello;

import org.oasisopen.sca.annotation.*;

@Remotable
public interface HelloService {

    String hello(String message);
}
```

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

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

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

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

The following snippet shows a remotable Java service interface.

```
package services.hello;

import org.oasisopen.sca.annotation.*;

@Remotable
public interface HelloService {

    String hello(String message);
}
```

```

2025 package services.hello;
2026
2027 import org.oasisopen.sca.annotation.*;
2028
2029 @Service(HelloService.class)
2030 public class HelloServiceImpl implements HelloService {
2031     public String hello(String message) {
2032         ...
2033     }
2034 }
2035

```

2036 9.19 @Requires

2037 The following Java code defines the *@Requires* annotation:

```

2038
2039 package org.oasisopen.sca.annotation;
2040
2041 import static java.lang.annotation.ElementType.FIELD;
2042 import static java.lang.annotation.ElementType.METHOD;
2043 import static java.lang.annotation.ElementType.PARAMETER;
2044 import static java.lang.annotation.ElementType.TYPE;
2045 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2046
2047 import java.lang.annotation.Inherited;
2048 import java.lang.annotation.Retention;
2049 import java.lang.annotation.Target;
2050
2051 @Inherited
2052 @Retention(RUNTIME)
2053 @Target({TYPE, METHOD, FIELD, PARAMETER})
2054 public @interface Requires {
2055     /**
2056      * Returns the attached intents.
2057      *
2058      * @return the attached intents
2059      */
2060     String[] value() default "";
2061 }
2062

```

2063 The *@Requires* annotation supports general purpose intents specified as strings. Users can also
2064 define specific intents using the *@Intent* annotation.

2065 See the [section "General Intent Annotations"](#) for details and samples.

2066 9.20 @Scope

2067 The following Java code defines the *@Scope* annotation:

```

2068 package org.oasisopen.sca.annotation;
2069
2070 import static java.lang.annotation.ElementType.TYPE;
2071 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2072 import java.lang.annotation.Retention;
2073 import java.lang.annotation.Target;
2074
2075 @Target(TYPE)
2076 @Retention(RUNTIME)

```

```
2077 public @interface Scope {
2078     String value() default "STATELESS";
2079 }
2080
```

2081 The @Scope annotation MUST only be used on a service's implementation class. It is an error to
2082 use this annotation on an interface.

2083 The @Scope annotation has the following attribute:

- 2084 • **value** – the name of the scope.
2085 For 'STATELESS' implementations, a different implementation instance can be used to
2086 service each request. Implementation instances can be newly created or be drawn from a
2087 pool of instances.
2088 SCA defines the following scope names, but others can be defined by particular Java-
2089 based implementation types:
2090 STATELESS
2091 COMPOSITE

2092 The default value is STATELESS.

2093 The following snippet shows a sample for a COMPOSITE scoped service implementation:

```
2094 package services.hello;
2095
2096 import org.oasisopen.sca.annotation.*;
2097
2098 @Service(HelloService.class)
2099 @Scope("COMPOSITE")
2100 public class HelloServiceImpl implements HelloService {
2101     public String hello(String message) {
2102         ...
2103     }
2104 }
2105
2106
```

2107 9.21 @Service

2108 The following Java code defines the @Service annotation:

```
2109 package org.oasisopen.sca.annotation;
2110
2111 import static java.lang.annotation.ElementType.TYPE;
2112 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2113 import java.lang.annotation.Retention;
2114 import java.lang.annotation.Target;
2115
2116 @Target(TYPE)
2117 @Retention(RUNTIME)
2118 public @interface Service {
2119     Class<?>[] interfaces() default {};
2120     Class<?> value() default Void.class;
2121 }
2122
2123
```

2124 The @Service annotation is used on a component implementation class to specify the SCA services
2125 offered by the implementation. The class need not be declared as implementing all of the
2126 interfaces implied by the services, but all methods of the service interfaces must be present. A
2127 class used as the implementation of a service is not required to have a @Service annotation. If a
2128 class has no @Service annotation, then the rules determining which services are offered and what
2129 interfaces those services have are determined by the specific implementation type.

2130 The @Service annotation has the following attributes:

- 2131 • **interfaces** – The value is an array of interface or class objects that should be exposed as
2132 services by this component.
- 2133 • **value** – A shortcut for the case when the class provides only a single service interface.

2134 Only one of these attributes should be specified.

2135

2136 A @Service annotation with no attributes is meaningless, it is the same as not having the
2137 annotation there at all.

2138 The **service names** of the defined services default to the names of the interfaces or class, without
2139 the package name.

2140 A component MUST NOT have two services with the same Java simple name. If a Java
2141 implementation needs to realize two services with the same Java simple name then this can be
2142 achieved through subclassing of the interface.

2143 The following snippet shows an implementation of the HelloService marked with the @Service
2144 annotation.

```
2145 package services.hello;  
2146  
2147 import org.oasisopen.sca.annotation.Service;  
2148  
2149 @Service(HelloService.class)  
2150 public class HelloServiceImpl implements HelloService {  
2151  
2152     public void hello(String name) {  
2153         System.out.println("Hello " + name);  
2154     }  
2155 }  
2156
```

2157

10 WSDL to Java and Java to WSDL

2158 The SCA Client and Implementation Model for Java applies the WSDL to Java and Java to WSDL
2159 mapping rules as defined by the JAX-WS specification [JAX-WS] for generating remotable Java
2160 interfaces from WSDL portTypes and vice versa.

2161 For the purposes of the Java-to-WSDL mapping algorithm, the interface is treated as if it had a
2162 @WebService annotation on the class, even if it doesn't, and the
2163 @org.oasisopen.sca.annotation.OneWay annotation should be treated as a synonym for the
2164 @javax.jws.OneWay annotation. For the WSDL-to-Java mapping, the generated @WebService
2165 annotation implies that the interface is @Remotable.

2166 For the mapping from Java types to XML schema types, SCA permits both the JAXB 2.1 [JAX-B]
2167 mapping and the SDO 2.1 [SDO] mapping. SCA runtimes MUST support the JAXB 2.1 mapping
2168 and MAY support the SDO 2.1 mapping. Having a choice of binding technologies is allowed, as
2169 noted in the first paragraph of section 5 of the JSR 181 (version 2) specification, which is
2170 referenced by the JAX-WS specification.

2171 The JAX-WS mappings are applied with the following restrictions:

- 2172 • No support for holders

2173

2174 **Note:** This specification needs more examples and discussion of how JAX-WS's client asynchronous
2175 model is used.

2176 10.1 JAX-WS Client Asynchronous API for a Synchronous Service

2177 The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client
2178 application with a means of invoking that service asynchronously, so that the client can invoke a service
2179 operation and proceed to do other work without waiting for the service operation to complete its
2180 processing. The client application can retrieve the results of the service either through a polling
2181 mechanism or via a callback method which is invoked when the operation completes.

2182 For SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional
2183 client-side asynchronous polling and callback methods defined by JAX-WS. For SCA service interfaces
2184 defined using interface.java, the Java interface MUST NOT contain these methods. If these methods are
2185 present, SCA Runtimes MUST NOT include them in the SCA reference interface as defined by the
2186 Assembly specification. These methods are recognized as follows.

2187 For each method M in the interface, if another method P in the interface has

- 2188 a. a method name that is M's method name with the characters "Async" appended, and
- 2189 b. the same parameter signature as M, and
- 2190 c. a return type of Response<R> where R is the return type of M

2191 then P is a JAX-WS polling method that isn't part of the SCA interface contract.

2192 For each method M in the interface, if another method C in the interface has

- 2193 a. a method name that is M's method name with the characters "Async" appended, and
- 2194 b. a parameter signature that is M's parameter signature with an additional final parameter of type
2195 AsyncHandler<R> where R is the return type of M, and
- 2196 c. a return type of Future<?>

2197 then C is a JAX-WS callback method that isn't part of the SCA interface contract.

2198 As an example, an interface can be defined in WSDL as follows:

2199
2200

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

```
2201 <part name="ticker" type="xsd:string"/>
2202 </message>
2203
2204 <message name="getPriceResponse">
2205 <part name="price" type="xsd:float"/>
2206 </message>
2207
2208 <portType name="StockQuote">
2209 <operation name="getPrice">
2210 <input message="tns:getPrice"/>
2211 <output message="tns:getPriceResponse"/>
2212 </operation>
2213 </portType>
```

2214

2215 The JAX-WS asynchronous mapping will produce the following Java interface:

```
2216 // asynchronous mapping
2217 @WebService
2218 public interface StockQuote {
2219     float getPrice(String ticker);
2220     Response<Float> getPriceAsync(String ticker);
2221     Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
2222 }
```

2223

2224 For SCA interface definition purposes, this is treated as equivalent to the following:

```
2225 // synchronous mapping
2226 @WebService
2227 public interface StockQuote {
2228     float getPrice(String ticker);
2229 }
```

2230

2231 SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above
2232 example, if the client implementation uses the asynchronous form of the interface, the two
2233 additional getPriceAsync() methods can be used for polling and callbacks as defined by the JAX-
2234 WS specification.

2235

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

```
2236 <?xml version="1.0" encoding="UTF-8"?>
2237 <!-- (c) Copyright SCA Collaboration 2006 -->
2238 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2239         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2240         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2241         elementFormDefault="qualified">
2242
2243     <include schemaLocation="sca-core.xsd"/>
2244
2245     <element name="interface.java" type="sca:JavaInterface"
2246             substitutionGroup="sca:interface"/>
2247     <complexType name="JavaInterface">
2248         <complexContent>
2249             <extension base="sca:Interface">
2250                 <sequence>
2251                     <any namespace="##other" processContents="lax"
2252                         minOccurs="0" maxOccurs="unbounded" />
2253                 </sequence>
2254                 <attribute name="interface" type="NCName" use="required"/>
2255                 <attribute name="callbackInterface" type="NCName"
2256                         use="optional"/>
2257                 <anyAttribute namespace="##any" processContents="lax" />
2258             </extension>
2259         </complexContent>
2260     </complexType>
2261 </schema>
```


2262

B. Conformance Items

2263 This section contains a list of conformance items for the SCA Java Common Annotations and APIs
2264 specification.

2265

Conformance ID	Description
[JCA30001]	@interface MUST be the fully qualified name of the Java interface class
[JCA30002]	@callbackInterface MUST be the fully qualified name of a Java interface used for callbacks
[JCA30003]	However, 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.

2266

2267

C. Acknowledgements

2268 The following individuals have participated in the creation of this specification and are gratefully
2269 acknowledged:

2270 **Participants:**

2271 [Participant Name, Affiliation | Individual Member]

2272 [Participant Name, Affiliation | Individual Member]

2273

D. Non-Normative Text

2275

E. Revision History

2276 [optional; should not be included in OASIS Standards]

2277

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

2278