



# Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

**Committee Draft 02**

**26 January 2009**

**Specification URIs:**

**This Version:**

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02.html>  
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02.doc>  
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02.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 Combella,	Avaya

**Editor(s):**

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

**Related work:**

This specification replaces or supersedes:

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

This specification is related to:

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

**Declared XML Namespace(s):**

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

**Abstract:**

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

Specifically, this specification covers:

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

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

**Status:**

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

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

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

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

---

## Notices

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

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

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

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

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

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

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

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

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

---

# Table of Contents

1	Introduction .....	7
1.1	Terminology .....	7
1.2	Normative References .....	7
1.3	Non-Normative References .....	8
2	Implementation Metadata .....	9
2.1	Service Metadata .....	9
2.1.1	@Service .....	9
2.1.2	Java Semantics of a Remotable Service .....	9
2.1.3	Java Semantics of a Local Service .....	9
2.1.4	@Reference .....	10
2.1.5	@Property .....	10
2.2	Implementation Scopes: @Scope, @Init, @Destroy .....	10
2.2.1	Stateless scope .....	11
2.2.2	Composite scope .....	11
2.2.3	Conversation scope .....	11
3	Interface .....	12
3.1	Java interface element – <interface.java> .....	12
3.2	@Remotable .....	13
3.3	@Conversational .....	13
3.4	@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 and Conversational Programming .....	16
6.1	@OneWay .....	16
6.2	Conversational Services .....	16
6.2.1	ConversationAttributes .....	16
6.2.2	@EndsConversation .....	17
6.3	Passing Conversational Services as Parameters .....	17
6.4	Conversational Client .....	17
6.5	Conversation Lifetime Summary .....	18
6.6	Conversation ID .....	19
6.6.1	Application Specified Conversation IDs .....	19
6.6.2	Accessing Conversation IDs from Clients .....	19
6.7	Callbacks .....	19
6.7.1	Stateful Callbacks .....	19
6.7.2	Stateless Callbacks .....	21
6.7.3	Implementing Multiple Bidirectional Interfaces .....	22
6.7.4	Accessing Callbacks .....	22
6.7.5	Customizing the Callback .....	23

6.7.6 Customizing the Callback Identity .....	23
6.7.7 Bindings for Conversations and Callbacks.....	24
7 Java API .....	25
7.1 Component Context.....	25
7.2 Request Context .....	26
7.3 CallableReference .....	27
7.4 ServiceReference .....	28
7.5 Conversation.....	28
7.6 ServiceRuntimeException.....	29
7.7 NoRegisteredCallbackException .....	29
7.8 ServiceUnavailableException .....	29
7.9 InvalidServiceException.....	29
7.10 ConversationEndedException .....	30
8 Java Annotations .....	31
8.1 @AllowsPassByReference .....	31
8.2 @Callback .....	32
8.3 @ComponentName .....	33
8.4 @Constructor.....	33
8.5 @Context.....	34
8.6 @Conversational .....	35
8.7 @ConversationAttributes.....	35
8.8 @ConversationID .....	36
8.9 @Destroy.....	37
8.10 @EagerInit.....	38
8.11 @EndsConversation.....	38
8.12 @Init.....	39
8.13 @OneWay .....	39
8.14 @Property.....	40
8.15 @Reference.....	41
8.15.1 Reinjection.....	44
8.16 @Remotable .....	45
8.17 @Scope .....	47
8.18 @Service .....	47
9 WSDL to Java and Java to WSDL .....	49
9.1 JAX-WS Client Asynchronous API for a Synchronous Service.....	49
10 Policy Annotations for Java .....	51
10.1 General Intent Annotations.....	51
10.2 Specific Intent Annotations .....	53
10.2.1 How to Create Specific Intent Annotations.....	54
10.3 Application of Intent Annotations .....	55
10.3.1 Inheritance And Annotation .....	56
10.4 Relationship of Declarative And Annotated Intents .....	57
10.5 Policy Set Annotations.....	58
10.6 Security Policy Annotations .....	58
10.6.1 Security Interaction Policy .....	58

10.6.2 Security Implementation Policy .....	61
A. XML Schema: sca-interface-java.xsd.....	65
B. Conformance Items .....	66
C. Acknowledgements .....	67
D. Non-Normative Text .....	68
E. Revision History.....	69

---

# 1 Introduction

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

Specifically, this specification covers:

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

Note that individual programming models may 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> , <a href="http://www.ietf.org/rfc/rfc2119.txt">http://www.ietf.org/rfc/rfc2119.txt</a> , IETF RFC 2119, March 1997.        |
| [ASSEMBLY] | SCA Assembly Specification, <a href="http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf">http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf</a> |
| [SDO]      | SDO 2.1 Specification, <a href="http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf">http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf</a>              |
| [JAX-B]    | JAXB 2.1 Specification, <a href="http://www.jcp.org/en/jsr/detail?id=222">http://www.jcp.org/en/jsr/detail?id=222</a>   |
| [WSDL]     | WSDL Specification,<br>WSDL 1.1: <a href="http://www.w3.org/TR/wsdl">http://www.w3.org/TR/wsdl</a> ,<br>WSDL 2.0: <a href="http://www.w3.org/TR/wsdl20/">http://www.w3.org/TR/wsdl20/</a>               |

43	<b>[POLICY]</b>	SCA Policy Framework, <a href="http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf">http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf</a>
44		
45	<b>[JSR-250]</b>	Common Annotation for Java Platform specification (JSR-250), <a href="http://www.jcp.org/en/jsr/detail?id=250">http://www.jcp.org/en/jsr/detail?id=250</a>
46		
47	<b>[JAX-WS]</b>	JAX-WS 2.1 Specification (JSR-224), <a href="http://www.jcp.org/en/jsr/detail?id=224">http://www.jcp.org/en/jsr/detail?id=224</a>
48		
49	<b>[JAVABEANS]</b>	JavaBeans 1.01 Specification, <a href="http://java.sun.com/javase/technologies/desktop/javabeans/api/">http://java.sun.com/javase/technologies/desktop/javabeans/api/</a>
50		
51		

## 52 **1.3 Non-Normative References**

53	<b>None</b>	None
----	-------------	------

---

## 54 2 Implementation Metadata

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

### 57 2.1 Service Metadata

#### 58 2.1.1 @Service

59

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

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

#### 67 2.1.2 Java Semantics of a Remotable Service

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

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

```
73 package services.hello;  
74 @Remotable  
75 public interface HelloService {  
76     String hello(String message);  
77 }  
78
```

#### 79 2.1.3 Java Semantics of a Local Service

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

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

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

85

```
86 package services.hello;  
87 public interface HelloService {  
88     String hello(String message);  
89 }  
90
```

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

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

#### 96 2.1.4 @Reference

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

#### 100 2.1.5 @Property

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

### 104 2.2 Implementation Scopes: @Scope, @Init, @Destroy

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

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

111 This document defines three scopes:

- 112 • STATELESS
- 113 • CONVERSATION
- 114 • COMPOSITE

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

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

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

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

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

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

```
127  
128     @Init  
129     public void start() {  
130         ...  
131     }  
132  
133     @Destroy  
134     public void stop() {  
135         ...
```

136        }  
137

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

## 140    2.2.1 Stateless scope

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

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

## 149    2.2.2 Composite scope

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

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

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

## 160    2.2.3 Conversation scope

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

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

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

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

---

## 178 3 Interface

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

### 180 3.1 Java interface element – <interface.java>

181 The Java interface element is used in SCDL files in places where an interface is declared in terms  
182 of a Java interface class. The Java interface element identifies the Java interface class and  
183 optionally identifies a callback interface, where the first Java interface represents the forward  
184 (service) call interface and the second interface represents the interface used to call back from the  
185 service to the client.

186  
187 The following is the pseudo-schema for the interface.java element

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

189  
190  
191 The interface.java element has the following attributes:

- 192 • **interface (1..1)** – the Java interface class to use for the service interface. @interface MUST  
193 be the fully qualified name of the Java interface class [JCA30001]
- 194 • **callbackInterface (0..1)** – the Java interface class to use for the callback interface.  
195 @callbackInterface MUST be the fully qualified name of a Java interface used for callbacks  
196 [JCA30002]

197  
198 The following snippet shows an example of the Java interface element:

```
199 <interface.java interface="services.stockquote.StockQuoteService"  
200 callbackInterface="services.stockquote.StockQuoteServiceCallback" />
```

201  
202  
203 Here, the Java interface is defined in the Java class file  
204 ./services/stockquote/StockQuoteService.class, where the root directory is defined by the  
205 contribution in which the interface exists. Similarly, the callback interface is defined in the Java  
206 class file ./services/stockquote/StockQuoteServiceCallback.class.

207 Note that the Java interface class identified by the @interface attribute can contain a Java  
208 @Callback annotation which identifies a callback interface. If this is the case, then it is not  
209 necessary to provide the @callbackInterface attribute. However, if the Java interface class  
210 identified by the @interface attribute does contain a Java @Callback annotation, then the Java  
211 interface class identified by the @callbackInterface attribute MUST be the same interface class.  
212 [JCA30003]

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

## 219 3.2 @Remotable

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

## 224 3.3 @Conversational

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

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

## 229 3.4 @Callback

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

---

## 234 4 Client API

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

### 237 4.1 Accessing Services from an SCA Component

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

#### 242 4.1.1 Using the Component Context API

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

### 246 4.2 Accessing Services from non-SCA component implementations

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

#### 249 4.2.1 ComponentContext

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

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

```
254  
255 ComponentContext context = // obtained through host environment-specific means  
256 HelloService helloService =  
257     context.getService(HelloService.class, "HelloService");  
258 String result = helloService.hello("Hello World!");
```

---

## 259 5 Error Handling

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

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

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

---

## 266 6 Asynchronous and Conversational Programming

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

- 274 • support for non-blocking method calls
- 275 • conversational services
- 276 • callbacks

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

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

### 282 6.1 @OneWay

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

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

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

### 293 6.2 Conversational Services

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

#### 297 6.2.1 ConversationAttributes

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

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

```
301 package com.bigbank;  
302 import org.oasisopen.sca.annotations.ConversationAttributes;  
303  
304 @ConversationAttributes(maxAge="30 days");  
305 public class LoanServiceImpl implements LoanService {  
306  
307 }
```

## 308 6.2.2 @EndsConversation

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

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

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

## 322 6.3 Passing Conversational Services as Parameters

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

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

```
328     interface ComponentContext{  
329         ...  
330         <B> ServiceReference<B> createSelfReference(Class  
331             businessInterface);  
332         <B> ServiceReference<B> createSelfReference(Class  
333             businessInterface, String serviceName);  
334     }
```

335

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

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

## 341 6.4 Conversational Client

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

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

349

```
350     @Reference  
351     LoanService loanService;  
352     // Known to be conversational because the interface is marked as  
353     // conversational
```

```

354     public void applyForMortgage(Customer customer, HouseInfo houseInfo,
355                                 int term)
356     {
357         LoanApplication loanApp;
358         loanApp = createApplication(customer, houseInfo);
359         loanService.apply(loanApp);
360         loanService.lockCurrentRate(term);
361     }
362
363     public boolean isApproved() {
364         return loanService.getLoanStatus().equals("approved");
365     }
366     public LoanApplication createApplication(Customer customer,
367                                             HouseInfo houseInfo) {
368         return ...;
369     }

```

## 370 6.5 Conversation Lifetime Summary

### 371 **Starting conversations**

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

- 373 • A @Reference to a conversational service is injected
- 374 • A call is made to CompositeContext.getServiceReference and then a method of the service
- 375 is called.
- 376

### 377 **Continuing conversations**

378 The client can continue an existing conversation, by:

- 379 • Holding the service reference that was created when the conversation started
- 380 • Getting the service reference object passed as a parameter from another service, even
- 381 remotely
- 382 • Loading a service reference that had been written to some form of persistent storage
- 383

### 384 **Ending conversations**

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

- 386 • A service operation that has been annotated @EndsConversation has been called
- 387 • The server calls an @EndsConversation method on the @Callback reference
- 388 • The server's conversation lifetime timeout occurs
- 389 • The client calls Conversation.end()
- 390 • Any non-business exception is thrown by a conversational operation
- 391

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

396 If a service reference is used after the service provider's conversation timeout has caused the  
397 conversation to be ended, then ConversationEndedException is thrown. In order to use that  
398 service reference for a new conversation, its endConversation () method must be called.  
399

## 400 6.6 Conversation ID

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

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

### 407 6.6.1 Application Specified Conversation IDs

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

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

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

### 416 6.6.2 Accessing Conversation IDs from Clients

417 Whether the conversation ID is chosen by the client or is generated by the system, the client may  
418 access the conversation ID by calling `getConversationID()` on the current conversation  
419 object.

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

## 423 6.7 Callbacks

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

- 428 • an interface for the provided service
- 429 • a callback interface that must be provided by the client

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

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

### 437 6.7.1 Stateful Callbacks

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

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

```
441 package somepackage;
442 import org.oasisopen.sca.annotations.Callback;
443 import org.oasisopen.sca.annotations.Conversational;
444 import org.oasisopen.sca.annotations.Remotable;
445 @Remotable
446 @Conversational
447 @Callback(MyServiceCallback.class)
448 public interface MyService {
449
450     void someMethod(String arg);
451 }
452
453 @Remotable
454 @Conversational
455 public interface MyServiceCallback {
456
457     void receiveResult(String result);
458 }
459
```

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

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

```
470 @Callback
471 protected MyServiceCallback callback;
472
473 @Reference
474 protected MyService backendService;
475
476 public void someMethod(String arg) {
477     backendService.someMethod(arg);
478 }
479
480 public void receiveResult(String result) {
481     callback.receiveResult(result);
482 }
483
```

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

489

```

490
491     private MyService myService;
492
493     @Reference
494     public void setMyService(MyService service) {
495         myService = service;
496     }
497
498     public void aClientMethod() {
499         ...
500         myService.someMethod(arg);
501     }
502
503     public void receiveResult(String result) {
504         // code to process the result
505     }
506

```

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

## 515 6.7.2 Stateless Callbacks

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

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

```

526
527     private ServiceReference<MyService> myService;
528
529     @Reference
530     public void setMyService(ServiceReference<MyService> service) {
531         myService = service;
532     }
533
534     public void aClientMethod() {
535         String someKey = "1234";
536         ...
537
538         myService.setCallbackID(someKey);
539         myService.getService().someMethod(arg);
540     }
541
542     @Context RequestContext context;
543
544     public void receiveResult(String result) {

```

```

545         Object key = context.getServiceReference().getCallbackID();
546         // Lookup any relevant state based on "key"
547         // code to process the result
548     }

```

549

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

```

552 @Callback
553 protected MyServiceCallback callback;

```

555

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

### 561 6.7.3 Implementing Multiple Bidirectional Interfaces

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

```

568 @Callback
569 protected MyService1Callback callback1;
570
571 @Callback
572 protected MyService2Callback callback2;

```

574

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

### 577 6.7.4 Accessing Callbacks

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

580

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

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

```

584 @Callback
585 protected CallableReference<MyCallback> callback;
586
587 public void someMethod() {
588     MyCallback myCallback = callback.getCallback();    ...
589     myCallback.receiveResult(theResult);
590 }

```

594

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

597

```
598 public void someMethod() {  
599     MyCallback myCallback =  
600         ComponentContext.getRequestContext().getCallback();  
601     ...  
602     ...  
603     myCallback.receiveResult(theResult);  
604 }  
605  
606  
607
```

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

610

```
611 @Context  
612 protected RequestContext requestContext;  
613  
614 void receiveResult(Object theResult) {  
615     Object refParams =  
616         requestContext.getServiceReference().getCallbackID();  
617     ...  
618 }  
619
```

620

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

## 625 6.7.5 Customizing the Callback

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

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

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

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

## 642 6.7.6 Customizing the Callback Identity

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

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

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

## 652 **6.7.7 Bindings for Conversations and Callbacks**

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

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

---

## 660 7 Java API

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

### 662 7.1 Component Context

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

```
664
665 package org.oasisopen.sca;
666
667 public interface ComponentContext {
668     String getURI();
669
670     <B> B getService(Class<B> businessInterface, String referenceName);
671
672     <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
673                                             String referenceName);
674
675     <B> Collection<B> getServices(Class<B> businessInterface,
676                                String referenceName);
677
678     <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
679                                                           businessInterface, String referenceName);
680
681     <B> ServiceReference<B> createSelfReference(Class<B>
682                                               businessInterface);
683
684     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
685                                               String serviceName);
686
687     <B> B getProperty(Class<B> type, String propertyName);
688
689     <B, R extends CallableReference<B>> R cast(B target)
690         throws IllegalArgumentException;
691
692     RequestContext getRequestContext();
693
694
695 }
```

696

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

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

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

728

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

```
731 private ComponentContext componentContext;
732
733 @Context
734 public void setContext(ComponentContext context) {
735     componentContext = context;
736 }
737
738 public void doSomething() {
739     HelloWorld service =
740     componentContext.getService(HelloWorld.class, "HelloWorldComponent");
741     service.hello("hello");
742 }
743
```

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

## 747 7.2 Request Context

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

749

```
750 package org.oasisopen.sca;
751
752 import javax.security.auth.Subject;
753
754 public interface RequestContext {
755
756     Subject getSecuritySubject();
757
```

```

758     String getServiceName();
759     <CB> CallableReference<CB> getCallbackReference();
760     <CB> CB getCallback();
761     <B> CallableReference<B> getServiceReference();
762
763 }
764

```

765 The RequestContext interface has the following methods:

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

## 779 7.3 CallableReference

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

```

781
782 package org.oasisopen.sca;
783
784 public interface CallableReference<B> extends java.io.Serializable {
785
786     B getService();
787     Class<B> getBusinessInterface();
788     boolean isConversational();
789     Conversation getConversation();
790     Object getCallbackID();
791 }
792

```

793 The CallableReference interface has the following methods:

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

## 805 7.4 ServiceReference

806

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

810 The following Java code defines the ServiceReference interface:

811

```
812 package org.oasisopen.sca;  
813  
814 public interface ServiceReference<B> extends CallableReference<B> {  
815     Object getConversationID();  
816     void setConversationID(Object conversationId) throws  
817         IllegalStateException;  
818     void setCallbackID(Object callbackID);  
819     Object getCallback();  
820     void setCallback(Object callback);  
821 }  
822
```

823

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

825

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

## 836 7.5 Conversation

837 The following snippet defines Conversation:

838

```
839 package org.oasisopen.sca;  
840  
841 public interface Conversation {  
842     Object getConversationID();  
843     void end();  
844 }
```

845

846 The Conversation interface has the following methods:

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

## 851 7.6 ServiceRuntimeException

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

853

```
854 package org.oasisopen.sca;  
855  
856 public class ServiceRuntimeException extends RuntimeException {  
857     ...  
858 }
```

859  
860 This exception signals problems in the management of SCA component execution.

## 861 7.7 NoRegisteredCallbackException

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

863

```
864 package org.oasisopen.sca;  
865  
866 public class NoRegisteredCallbackException extends  
867     ServiceRuntimeException {  
868     ...  
869 }
```

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

## 873 7.8 ServiceUnavailableException

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

875

```
876 package org.oasisopen.sca;  
877  
878 public class ServiceUnavailableException extends ServiceRuntimeException {  
879     ...  
880 }
```

881

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

## 886 7.9 InvalidServiceException

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

888

```
889 package org.oasisopen.sca;  
890  
891 public class InvalidServiceException extends ServiceRuntimeException {  
892     ...  
893 }
```

894

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

## 898 7.10 ConversationEndedException

899 The following snippet shows the *ConversationEndedException*.

```
900  
901 package org.oasisopen.sca;  
902  
903 public class ConversationEndedException extends ServiceRuntimeException {  
904     ...  
905 }  
906
```

---

## 8 Java Annotations

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

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

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

### 8.1 @AllowsPassByReference

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

```
package org.oasisopen.sca.annotations;

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

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

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

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

```
@AllowsPassByReference
public String hello(String message) {
    ...
}
```

## 952 8.2 @Callback

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

954

```
955 package org.oasisopen.sca.annotations;
956
957 import static java.lang.annotation.ElementType.TYPE;
958 import static java.lang.annotation.ElementType.METHOD;
959 import static java.lang.annotation.ElementType.FIELD;
960 import static java.lang.annotation.RetentionPolicy.RUNTIME;
961 import java.lang.annotation.Retention;
962 import java.lang.annotation.Target;
963
964 @Target(TYPE, METHOD, FIELD)
965 @Retention(RUNTIME)
966 public @interface Callback {
967
968     Class<?> value() default Void.class;
969 }
970
971
```

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

974 The @Callback annotation has the following attribute:

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

976

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

979

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

981

```
982 @Remotable
983 @Callback(MyServiceCallback.class)
984 public interface MyService {
985
986     void someAsyncMethod(String arg);
987 }
988
```

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

990

```
991 package somepackage;
992 import org.oasisopen.sca.annotations.Callback;
993 import org.oasisopen.sca.annotations.Remotable;
994 @Remotable
995 @Callback(MyServiceCallback.class)
996 public interface MyService {
997
998     void someMethod(String arg);
999 }
1000
1001 @Remotable
1002 public interface MyServiceCallback {
```

```
1003
1004     void receiveResult(String result);
1005 }
```

1006

1007 In this example, the implied component type is:

1008

```
1009 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >
1010     <service name="MyService">
1011         <interface.java interface="somepackage.MyService"
1012             callbackInterface="somepackage.MyServiceCallback"/>
1013     </service>
1014 </componentType>
```

### 1016 8.3 @ComponentName

1017 The following Java code defines the **@ComponentName** annotation:

1018

```
1019 package org.oasisopen.sca.annotations;
1020
1021 import static java.lang.annotation.ElementType.METHOD;
1022 import static java.lang.annotation.ElementType.FIELD;
1023 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1024 import java.lang.annotation.Retention;
1025 import java.lang.annotation.Target;
1026
1027 @Target({METHOD, FIELD})
1028 @Retention(RUNTIME)
1029 public @interface ComponentName {
1030
1031 }
1032
```

1033 The @ComponentName annotation is used to denote a Java class field or setter method that is  
1034 used to inject the component name.

1035

1036 The following snippet shows a component name field definition sample.

1037

```
1038 @ComponentName
1039 private String componentName;
```

1041 The following snippet shows a component name setter method sample.

1042

```
1043 @ComponentName
1044 public void setComponentName(String name) {
1045     //...
1046 }
```

### 1047 8.4 @Constructor

1048 The following Java code defines the **@Constructor** annotation:

1049

```

1050 package org.oasisopen.sca.annotations;
1051
1052 import static java.lang.annotation.ElementType.CONSTRUCTOR;
1053 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1054 import java.lang.annotation.Retention;
1055 import java.lang.annotation.Target;
1056
1057 @Target( CONSTRUCTOR )
1058 @Retention( RUNTIME )
1059 public @interface Constructor { }
1060

```

1061 The @Constructor annotation is used to mark a particular constructor to use when instantiating a  
1062 Java component implementation. If this constructor has parameters, each of these parameters  
1063 MUST have either a @Property annotation or a @Reference annotation.

1064 The following snippet shows a sample for the @Constructor annotation.

```

1065
1066 public class HelloServiceImpl implements HelloService {
1067
1068     public HelloServiceImpl(){
1069         ...
1070     }
1071
1072     @Constructor
1073     public HelloServiceImpl(@Property(name="someProperty") String
1074 someProperty ){
1075         ...
1076     }
1077
1078     public String hello(String message) {
1079         ...
1080     }
1081 }

```

## 1082 8.5 @Context

1083 The following Java code defines the **@Context** annotation:

```

1084
1085 package org.oasisopen.sca.annotations;
1086
1087 import static java.lang.annotation.ElementType.METHOD;
1088 import static java.lang.annotation.ElementType.FIELD;
1089 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1090 import java.lang.annotation.Retention;
1091 import java.lang.annotation.Target;
1092
1093 @Target( {METHOD, FIELD} )
1094 @Retention( RUNTIME )
1095 public @interface Context {
1096
1097 }
1098

```

1099 The @Context annotation is used to denote a Java class field or a setter method that is used to  
1100 inject a composite context for the component. The type of context to be injected is defined by the

1101 type of the Java class field or type of the setter method input argument; the type is either  
1102 **ComponentContext** or **RequestContext**.

1103 The @Context annotation has no attributes.

1104

1105 The following snippet shows a ComponentContext field definition sample.

1106

```
1107 @Context  
1108 protected ComponentContext context;
```

1109

1110 The following snippet shows a RequestContext field definition sample.

1111

```
1112 @Context  
1113 protected RequestContext context;
```

## 1114 8.6 @Conversational

1115 The following Java code defines the **@Conversational** annotation:

1116

```
1117 package org.oasisopen.sca.annotations;  
1118  
1119 import static java.lang.annotation.ElementType.TYPE;  
1120 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1121 import java.lang.annotation.Retention;  
1122 import java.lang.annotation.Target;  
1123 @Target(TYPE)  
1124 @Retention(RUNTIME)  
1125 public interface Conversational {  
1126 }  
1127
```

1128 The @Conversational annotation is used on a Java interface to denote a conversational service  
1129 contract.

1130 The @Conversational annotation has no attributes.

1131 The following snippet shows a sample for the @Conversational annotation.

```
1132 package services.hello;  
1133  
1134 import org.oasisopen.sca.annotations.Conversational;  
1135  
1136 @Conversational  
1137 public interface HelloService {  
1138     void setName(String name);  
1139     String sayHello();  
1140 }
```

## 1141 8.7 @ConversationAttributes

1142 The following Java code defines the **@ConversationAttributes** annotation:

1143

```
1144 package org.oasisopen.sca.annotations;  
1145  
1146 import static java.lang.annotation.ElementType.TYPE;  
1147 import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```

1148 import java.lang.annotation.Retention;
1149 import java.lang.annotation.Target;
1150
1151 @Target(TYPE)
1152 @Retention(RUNTIME)
1153 public @interface ConversationAttributes {
1154
1155     String maxIdleTime() default "";
1156     String maxAge() default "";
1157     boolean singlePrincipal() default false;
1158 }
1159

```

1160 The @ConversationAttributes annotation is used to define a set of attributes which apply to  
1161 conversational interfaces of services or references of a Java class. The annotation has the following  
1162 attributes:

- 1163 • **maxIdleTime (optional)** - The maximum time that can pass between successive  
1164 operations within a single conversation. If more time than this passes, then the container  
1165 may end the conversation.
- 1166 • **maxAge (optional)** - The maximum time that the entire conversation can remain active.  
1167 If more time than this passes, then the container may end the conversation.
- 1168 • **singlePrincipal (optional)** – If true, only the principal (the user) that started the  
1169 conversation has authority to continue the conversation. The default value is false.

1170

1171 The two attributes that take a time express the time as a string that starts with an integer, is  
1172 followed by a space and then one of the following: "seconds", "minutes", "hours", "days" or  
1173 "years".

1174

1175 Not specifying timeouts means that timeouts are defined by the SCA runtime implementation,  
1176 however it chooses to do so.

1177

1178 The following snippet shows the use of the @ConversationAttributes annotation to set the  
1179 maximum age for a Conversation to be 30 days.

1180

```

1181 package service.shoppingcart;
1182
1183 import org.oasisopen.sca.annotations.ConversationAttributes;
1184
1185 @ConversationAttributes (maxAge="30 days");
1186 public class ShoppingCartServiceImpl implements ShoppingCartService {
1187     ...
1188 }

```

## 1189 8.8 @ConversationID

1190 The following Java code defines the @ConversationID annotation:

1191

```

1192 package org.oasisopen.sca.annotations;
1193
1194 import static java.lang.annotation.ElementType.METHOD;
1195 import static java.lang.annotation.ElementType.FIELD;
1196 import static java.lang.annotation.RetentionPolicy.RUNTIME;

```

```

1197     import java.lang.annotation.Retention;
1198     import java.lang.annotation.Target;
1199
1200     @Target({METHOD, FIELD})
1201     @Retention(RUNTIME)
1202     public @interface ConversationID {
1203
1204     }
1205

```

1206 The @ConversationID annotation is used to annotate a Java class field or setter method that is  
1207 used to inject the conversation ID. System generated conversation IDs are always strings, but  
1208 application generated conversation IDs may be other complex types.

1209 The following snippet shows a conversation ID field definition sample.

```

1210
1211     @ConversationID
1212     private String conversationID;
1213

```

1214 The type of the field is not necessarily String.

1215

## 1216 8.9 @Destroy

1217 The following Java code defines the **@Destroy** annotation:

1218

```

1219     package org.oasisopen.sca.annotations;
1220
1221     import static java.lang.annotation.ElementType.METHOD;
1222     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1223     import java.lang.annotation.Retention;
1224     import java.lang.annotation.Target;
1225
1226     @Target(METHOD)
1227     @Retention(RUNTIME)
1228     public @interface Destroy {
1229
1230     }
1231

```

1232 The @Destroy annotation is used to denote a single Java class method that will be called when the  
1233 scope defined for the implementation class ends. The method MAY have any access modifier and  
1234 MUST have a void return type and no arguments.

1235 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method  
1236 when the scope defined for the implementation class ends. If the implementation class has a  
1237 method with an @Destroy annotation that does not match these criteria, the SCA runtime MUST  
1238 NOT instantiate the implementation class.

1239

1240 The following snippet shows a sample for a destroy method definition.

1241

```

1242     @Destroy
1243     public void myDestroyMethod() {
1244         ...
1245     }

```

## 1246 8.10 @EagerInit

1247 The following Java code defines the **@EagerInit** annotation:

1248

```
1249 package org.oasisopen.sca.annotations;
1250
1251 import static java.lang.annotation.ElementType.TYPE;
1252 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1253 import java.lang.annotation.Retention;
1254 import java.lang.annotation.Target;
1255
1256 @Target (TYPE)
1257 @Retention(RUNTIME)
1258 public @interface EagerInit {
1259
1260 }
1261
```

1262 The **@EagerInit** annotation is used to annotate the Java class of a COMPOSITE scoped  
1263 implementation for eager initialization. When marked for eager initialization, the composite scoped  
1264 instance is created when its containing component is started.

## 1265 8.11 @EndsConversation

1266 The following Java code defines the **@EndsConversation** annotation:

1267

```
1268 package org.oasisopen.sca.annotations;
1269
1270 import static java.lang.annotation.ElementType.METHOD;
1271 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1272 import java.lang.annotation.Retention;
1273 import java.lang.annotation.Target;
1274
1275 @Target (METHOD)
1276 @Retention(RUNTIME)
1277 public @interface EndsConversation {
1278
1279 }
1280
1281
```

1282 The @EndsConversation annotation is used to denote a method on a Java interface that is called  
1283 to end a conversation.

1284 The @EndsConversation annotation has no attributes.

1285 The following snippet shows a sample using the @EndsConversation annotation.

```
1286 package services.shoppingbasket;
1287
1288 import org.oasisopen.sca.annotations.EndsConversation;
1289
1290 public interface ShoppingBasket {
1291     void addItem(String itemID, int quantity);
1292
1293     @EndsConversation
1294     void buy();
1295 }
```

## 1296 8.12 @Init

1297 The following Java code defines the **@Init** annotation:

1298

```
1299 package org.oasisopen.sca.annotations;  
1300  
1301 import static java.lang.annotation.ElementType.METHOD;  
1302 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1303 import java.lang.annotation.Retention;  
1304 import java.lang.annotation.Target;  
1305  
1306 @Target(METHOD)  
1307 @Retention(RUNTIME)  
1308 public @interface Init {  
1309  
1310 }  
1311  
1312
```

1313 The @Init annotation is used to denote a single Java class method that is called when the scope  
1314 defined for the implementation class starts. The method MAY have any access modifier and MUST  
1315 have a void return type and no arguments.

1316 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method  
1317 after all property and reference injection is complete. If the implementation class has a method  
1318 with an @Init annotation that does not match these criteria, the SCA runtime MUST NOT  
1319 instantiate the implementation class.

1320 The following snippet shows an example of an init method definition.

1321

```
1322 @Init  
1323 public void myInitMethod() {  
1324     ...  
1325 }
```

## 1326 8.13 @OneWay

1327 The following Java code defines the **@OneWay** annotation:

1328

```
1329 package org.oasisopen.sca.annotations;  
1330  
1331 import static java.lang.annotation.ElementType.METHOD;  
1332 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1333 import java.lang.annotation.Retention;  
1334 import java.lang.annotation.Target;  
1335  
1336 @Target(METHOD)  
1337 @Retention(RUNTIME)  
1338 public @interface OneWay {  
1339  
1340 }  
1341  
1342
```

1343 The @OneWay annotation is used on a Java interface or class method to indicate that invocations  
1344 will be dispatched in a non-blocking fashion as described in the section on Asynchronous  
1345 Programming.

1346 The @OneWay annotation has no attributes.  
1347 The following snippet shows the use of the @OneWay annotation on an interface.

```
1348 package services.hello;  
1349  
1350 import org.oasisopen.sca.annotations.OneWay;  
1351  
1352 public interface HelloService {  
1353     @OneWay  
1354     void hello(String name);  
1355 }
```

## 1356 8.14 @Property

1357 The following Java code defines the **@Property** annotation:

```
1358  
1359 package org.oasisopen.sca.annotations;  
1360  
1361 import static java.lang.annotation.ElementType.METHOD;  
1362 import static java.lang.annotation.ElementType.FIELD;  
1363 import static java.lang.annotation.ElementType.PARAMETER;  
1364 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1365 import java.lang.annotation.Retention;  
1366 import java.lang.annotation.Target;  
1367  
1368 @Target({METHOD, FIELD, PARAMETER})  
1369 @Retention(RUNTIME)  
1370 public @interface Property {  
1371  
1372     String name() default "";  
1373     boolean required() default true;  
1374 }  
1375
```

1376  
1377 The @Property annotation is used to denote a Java class field, a setter method, or a constructor  
1378 parameter that is used to inject an SCA property value. The type of the property injected, which  
1379 can be a simple Java type or a complex Java type, is defined by the type of the Java class field or  
1380 the type of the input parameter of the setter method or constructor.

1381 The @Property annotation may be used on fields, on setter methods or on a constructor method  
1382 parameter. However, the @Property annotation MUST NOT be used on a class field that is declared  
1383 as final.

1384 Properties may also be injected via setter methods even when the @Property annotation is not  
1385 present. However, the @Property annotation must be used in order to inject a property onto a  
1386 non-public field. In the case where there is no @Property annotation, the name of the property is  
1387 the same as the name of the field or setter.

1388 Where there is both a setter method and a field for a property, the setter method is used.

1389  
1390 The @Property annotation has the following attributes:

- 1391 • **name (optional)** – the name of the property. For a field annotation, the default is the  
1392 name of the field of the Java class. For a setter method annotation, the default is the  
1393 JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a  
1394 constructor parameter annotation, there is no default and the name attribute MUST be  
1395 present.

- 1396           • **required (optional)** – specifies whether injection is required, defaults to true. For a  
1397           constructor parameter annotation, this attribute **MUST** have the value true.

1398

1399       The following snippet shows a property field definition sample.

1400

```
1401       @property(name="currency", required=true)  
1402       protected String currency;
```

1403

1404       The following snippet shows a property setter sample

1405

```
1406       @property(name="currency", required=true)  
1407       public void setCurrency( String theCurrency ) {  
1408               ....  
1409       }
```

1410

1411       If the property is defined as an array or as any type that extends or implements  
1412       **java.util.Collection**, then the implied component type has a property with a **many** attribute set to  
1413       true.

1414

1415       The following snippet shows the definition of a configuration property using the @Property  
1416       annotation for a collection.

1417

```
1418       ...  
1419       private List<String> helloConfigurationProperty;  
1420  
1421       @property(required=true)  
1422       public void setHelloConfigurationProperty(List<String> property) {  
1423               helloConfigurationProperty = property;  
1424       }  
1425       ...
```

## 1426   8.15 @Reference

1427       The following Java code defines the **@Reference** annotation:

1428

```
1429       package org.oasisopen.sca.annotations;  
1430  
1431       import static java.lang.annotation.ElementType.METHOD;  
1432       import static java.lang.annotation.ElementType.FIELD;  
1433       import static java.lang.annotation.ElementType.PARAMETER;  
1434       import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1435       import java.lang.annotation.Retention;  
1436       import java.lang.annotation.Target;  
1437       @Target({METHOD, FIELD, PARAMETER})  
1438       @Retention(RUNTIME)  
1439       public @interface Reference {  
1440  
1441               String name() default "";  
1442               boolean required() default true;
```

1443 }  
1444

1445 The @Reference annotation type is used to annotate a Java class field, a setter method, or a  
1446 constructor parameter that is used to inject a service that resolves the reference. The interface of  
1447 the service injected is defined by the type of the Java class field or the type of the input parameter  
1448 of the setter method or constructor.

1449 The @Reference annotation MUST NOT be used on a class field that is declared as final.

1450 References may also be injected via setter methods even when the @Reference annotation is not  
1451 present. However, the @Reference annotation must be used in order to inject a reference onto a  
1452 non-public field. In the case where there is no @Reference annotation, the name of the reference  
1453 is the same as the name of the field or setter.

1454 Where there is both a setter method and a field for a reference, the setter method is used.

1455 The @Reference annotation has the following attributes:

- 1456 • **name (optional)** – the name of the reference. For a field annotation, the default is the  
1457 name of the field of the Java class. For a setter method annotation, the default is the  
1458 JavaBeans property name corresponding to the setter method name. For a constructor  
1459 parameter annotation, there is no default and the name attribute MUST be present.
- 1460 • **required (optional)** – whether injection of service or services is required. Defaults to true.  
1461 For a constructor parameter annotation, this attribute MUST have the value true.

1462

1463 The following snippet shows a reference field definition sample.

1464

```
1465 @Reference(name="stockQuote", required=true)  
1466 protected StockQuoteService stockQuote;
```

1467

1468 The following snippet shows a reference setter sample

1469

```
1470 @Reference(name="stockQuote", required=true)  
1471 public void setStockQuote( StockQuoteService theSQService ) {  
1472     ...  
1473 }
```

1474

1475 The following fragment from a component implementation shows a sample of a service reference  
1476 using the @Reference annotation. The name of the reference is "helloService" and its type is  
1477 HelloService. The clientMethod() calls the "hello" operation of the service referenced by the  
1478 helloService reference.

1479

```
1480 package services.hello;  
1481  
1482 private HelloService helloService;  
1483  
1484 @Reference(name="helloService", required=true)  
1485 public setHelloService(HelloService service) {  
1486     helloService = service;  
1487 }  
1488  
1489 public void clientMethod() {  
1490     String result = helloService.hello("Hello World!");
```

1491  
1492  
1493

```
    ...  
}
```

1494 The presence of a @Reference annotation is reflected in the componentType information that the  
1495 runtime generates through reflection on the implementation class. The following snippet shows  
1496 the component type for the above component implementation fragment.

1497

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

1508 If the reference is not an array or collection, then the implied component type has a reference  
1509 with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference **required**  
1510 attribute – 1..1 applies if required=true.

1511

1512 If the reference is defined as an array or as any type that extends or implements **java.util.Collection**,  
1513 then the implied component type has a reference with a **multiplicity** of either **1..n** or **0..n**, depending  
1514 on whether the **required** attribute of the @Reference annotation is set to true or false – 1..n applies if  
1515 required=true.

1516

1517 The following fragment from a component implementation shows a sample of a service reference  
1518 definition using the @Reference annotation on a java.util.List. The name of the reference is  
1519 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the  
1520 services referenced by the helloServices reference. In this case, at least one HelloService should  
1521 be present, so **required** is true.

1522

```
@Reference(name="helloServices", required=true)  
protected List<HelloService> helloServices;  
  
public void clientMethod() {  
    ...  
    for (int index = 0; index < helloServices.size(); index++) {  
        HelloService helloService =  
            (HelloService)helloServices.get(index);  
        String result = helloService.hello("Hello World!");  
    }  
    ...  
}
```

1537 The following snippet shows the XML representation of the component type reflected from for the  
1538 former component implementation fragment. There is no need to author this component type in  
1539 this case since it can be reflected from the Java class.

1540

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

```
1544     <!-- Any services offered by the component would be listed here -->
1545     <reference name="helloServices" multiplicity="1..n">
1546         <interface.java interface="services.hello.HelloService"/>
1547     </reference>
1548
1549 </componentType>
```

1550  
1551 At runtime, the representation of an unwired reference depends on the reference's multiplicity. An  
1552 unwired reference with a multiplicity of 0..1 must be null. An unwired reference with a multiplicity  
1553 of 0..N must be an empty array or collection.

### 1554 8.15.1 Reinjection

1555 References MAY be reinjected after the initial creation of a component if the reference target  
1556 changes due to a change in wiring that has occurred since the component was initialized. In order  
1557 for reinjection to occur, the following MUST be true:

- 1558 1. The component MUST NOT be STATELESS scoped.
- 1559 2. The reference MUST use either field-based injection or setter injection. References that are  
1560 injected through constructor injection MUST NOT be changed. Setter injection allows for  
1561 code in the setter method to perform processing in reaction to a change.
- 1562 3. If the reference has a conversational interface, then reinjection MUST NOT occur while the  
1563 conversation is active.

1564 If a reference target changes and the reference is not reinjected, the reference MUST continue to  
1565 work as if the reference target was not changed.

1566 If an operation is called on a reference where the target of that reference has been undeployed,  
1567 the SCA runtime SHOULD throw `InvalidServiceException`. If an operation is called on a reference  
1568 where the target of the reference has become unavailable for some reason, the SCA runtime  
1569 SHOULD throw `ServiceUnavailableException`. If the target of the reference is changed, the  
1570 reference MAY continue to work, depending on the runtime and the type of change that was made.  
1571 If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.

1572 A `ServiceReference` that has been obtained from a reference by `ComponentContext.cast()`  
1573 corresponds to the reference that is passed as a parameter to `cast()`. If the reference is  
1574 subsequently reinjected, the `ServiceReference` obtained from the original reference MUST continue  
1575 to work as if the reference target was not changed. If the target of a `ServiceReference` has been  
1576 undeployed, the SCA runtime SHOULD throw `InvalidServiceException` when an operation is  
1577 invoked on the `ServiceReference`. If the target of a `ServiceReference` has become unavailable, the  
1578 SCA runtime SHOULD throw `ServiceUnavailableException` when an operation is invoked on the  
1579 `ServiceReference`. If the target of a `ServiceReference` is changed, the reference MAY continue to  
1580 work, depending on the runtime and the type of change that was made. If it doesn't work, the  
1581 exception thrown will depend on the runtime and the cause of the failure.

1582 A reference or `ServiceReference` accessed through the component context by calling `getService()`  
1583 or `getServiceReference()` MUST correspond to the current configuration of the domain. This  
1584 applies whether or not reinjection has taken place. If the target has been undeployed or has  
1585 become unavailable, the result SHOULD be a reference to the undeployed or unavailable service,  
1586 and attempts to call business methods SHOULD throw an exception as described above. If the  
1587 target has changed, the result SHOULD be a reference to the changed service.

1588 The rules for reference reinjection also apply to references with a multiplicity of 0..N or 1..N. This  
1589 means that in the cases listed above where reference reinjection is not allowed, the array or  
1590 Collection for the reference MUST NOT change its contents. In cases where the contents of a  
1591 reference collection MAY change, then for references that use setter injection, the setter method  
1592 MUST be called for any change to the contents. The reinjected array or Collection MUST NOT be  
1593 the same array or Collection object previously injected to the component.

1594

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

1595

## 1596 8.16 @Remotable

1597 The following Java code defines the **@Remotable** annotation:

1598

```
1599 package org.oasisopen.sca.annotations;
```

```
1600
```

```
1601 import static java.lang.annotation.ElementType.TYPE;
```

```
1602 import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```
1603 import java.lang.annotation.Retention;
```

```
1604 import java.lang.annotation.Target;
```

```
1605
```

```
1606
```

```
1607 @Target (TYPE)
```

```
1608 @Retention(RUNTIME)
```

```
1609 public @interface Remotable {
```

```
1610
```

```
1611 }
```

```
1612
```

1613 The @Remotable annotation is used to specify a Java service interface as remotable. A remotable  
1614 service can be published externally as a service and must be translatable into a WSDL portType.

1615 The @Remotable annotation has no attributes.

1616

1617 The following snippet shows the Java interface for a remotable service with its @Remotable  
1618 annotation.

```
1619 package services.hello;
1620
1621 import org.oasisopen.sca.annotations.*;
1622
1623 @Remotable
1624 public interface HelloService {
1625     String hello(String message);
1626 }
1627
1628
```

1629 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**  
1630 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

1631

1632 Complex data types exchanged via remotable service interfaces MUST be compatible with the  
1633 marshalling technology used by the service binding. For example, if the service is going to be  
1634 exposed using the standard Web Service binding, then the parameters MAY be JAXB [JAX-B] types  
1635 or Service Data Objects (SDOs) [SDO].

1636 Independent of whether the remotable service is called from outside of the composite that  
1637 contains it or from another component in the same composite, the data exchange semantics are  
1638 **by-value**.

1639 Implementations of remotable services may modify input data during or after an invocation and  
1640 may modify return data after the invocation. If a remotable service is called locally or remotely,  
1641 the SCA container is responsible for making sure that no modification of input data or post-  
1642 invocation modifications to return data are seen by the caller.

1643

1644 The following snippet shows a remotable Java service interface.

1645

```
1646 package services.hello;
1647
1648 import org.oasisopen.sca.annotations.*;
1649
1650 @Remotable
1651 public interface HelloService {
1652     String hello(String message);
1653 }
1654
1655 package services.hello;
1656
1657 import org.oasisopen.sca.annotations.*;
1658
1659 @Service(HelloService.class)
1660 public class HelloServiceImpl implements HelloService {
1661     public String hello(String message) {
1662         ...
1663     }
1664
```

```
1665     }
1666 }
```

## 1667 8.17 @Scope

1668 The following Java code defines the **@Scope** annotation:

```
1669
1670 package org.oasisopen.sca.annotations;
1671
1672 import static java.lang.annotation.ElementType.TYPE;
1673 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1674 import java.lang.annotation.Retention;
1675 import java.lang.annotation.Target;
1676
1677 @Target(TYPE)
1678 @Retention(RUNTIME)
1679 public @interface Scope {
1680
1681     String value() default "STATELESS";
1682 }
```

1683 The @Scope annotation may only be used on a service's implementation class. It is an error to use  
1684 this annotation on an interface.

1685 The @Scope annotation has the following attribute:

- 1686 • **value** – the name of the scope.  
1687 For 'STATELESS' implementations, a different implementation instance may be used to  
1688 service each request. Implementation instances may be newly created or be drawn from a  
1689 pool of instances.  
1690 SCA defines the following scope names, but others can be defined by particular Java-  
1691 based implementation types:  
1692 STATELESS  
1693 COMPOSITE  
1694 CONVERSATION

1695 The default value is STATELESS, except for an implementation offering a @Conversational service,  
1696 which has a default scope of CONVERSATION. See section 2.2 for more details of the SCA-defined  
1697 scopes.

1698 The following snippet shows a sample for a CONVERSATION scoped service implementation:

```
1699 package services.hello;
1700
1701 import org.oasisopen.sca.annotations.*;
1702
1703 @Service(HelloService.class)
1704 @Scope("CONVERSATION")
1705 public class HelloServiceImpl implements HelloService {
1706
1707     public String hello(String message) {
1708         ...
1709     }
1710 }
1711
```

## 1712 8.18 @Service

1713 The following Java code defines the **@Service** annotation:

1714

```

1715 package org.oasisopen.sca.annotations;
1716
1717 import static java.lang.annotation.ElementType.TYPE;
1718 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1719 import java.lang.annotation.Retention;
1720 import java.lang.annotation.Target;
1721
1722 @Target(TYPE)
1723 @Retention(RUNTIME)
1724 public @interface Service {
1725
1726     Class<?>[] interfaces() default {};
1727     Class<?> value() default Void.class;
1728 }
1729

```

1730 The @Service annotation is used on a component implementation class to specify the SCA services  
1731 offered by the implementation. The class need not be declared as implementing all of the  
1732 interfaces implied by the services, but all methods of the service interfaces must be present. A  
1733 class used as the implementation of a service is not required to have a @Service annotation. If a  
1734 class has no @Service annotation, then the rules determining which services are offered and what  
1735 interfaces those services have are determined by the specific implementation type.

1736 The @Service annotation has the following attributes:

- 1737 • **interfaces** – The value is an array of interface or class objects that should be exposed as  
1738 services by this component.
- 1739 • **value** – A shortcut for the case when the class provides only a single service interface.

1740 Only one of these attributes should be specified.

1741

1742 A @Service annotation with no attributes is meaningless, it is the same as not having the  
1743 annotation there at all.

1744 The **service names** of the defined services default to the names of the interfaces or class, without  
1745 the package name.

1746 A component MUST NOT have two services with the same Java simple name. If a Java  
1747 implementation needs to realize two services with the same Java simple name then this can be  
1748 achieved through subclassing of the interface.

1749 The following snippet shows an implementation of the HelloService marked with the @Service  
1750 annotation.

```

1751 package services.hello;
1752
1753 import org.oasisopen.sca.annotations.Service;
1754
1755 @Service(HelloService.class)
1756 public class HelloServiceImpl implements HelloService {
1757
1758     public void hello(String name) {
1759         System.out.println("Hello " + name);
1760     }
1761 }
1762

```

1763

## 9 WSDL to Java and Java to WSDL

1764 The SCA Client and Implementation Model for Java applies the WSDL to Java and Java to WSDL  
1765 mapping rules as defined by the JAX-WS specification [JAX-WS] for generating remotable Java  
1766 interfaces from WSDL portTypes and vice versa.

1767 For the purposes of the Java-to-WSDL mapping algorithm, the interface is treated as if it had a  
1768 @WebService annotation on the class, even if it doesn't, and the  
1769 @org.oasisopen.annotations.OneWay annotation should be treated as a synonym for the  
1770 @javax.jws.OneWay annotation. For the WSDL-to-Java mapping, the generated @WebService  
1771 annotation implies that the interface is @Remotable.

1772 For the mapping from Java types to XML schema types, SCA permits both the JAXB 2.1 [JAX-B]  
1773 mapping and the SDO 2.1 [SDO] mapping. SCA runtimes MUST support the JAXB 2.1 mapping  
1774 and MAY support the SDO 2.1 mapping. Having a choice of binding technologies is allowed, as  
1775 noted in the first paragraph of section 5 of the JSR 181 (version 2) specification, which is  
1776 referenced by the JAX-WS specification.

1777 The JAX-WS mappings are applied with the following restrictions:

- 1778 • No support for holders

1779

1780 **Note:** This specification needs more examples and discussion of how JAX-WS's client asynchronous  
1781 model is used.

### 9.1 JAX-WS Client Asynchronous API for a Synchronous Service

1782 The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client  
1783 application with a means of invoking that service asynchronously, so that the client can invoke a service  
1784 operation and proceed to do other work without waiting for the service operation to complete its  
1785 processing. The client application can retrieve the results of the service either through a polling  
1786 mechanism or via a callback method which is invoked when the operation completes.

1788 For SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional  
1789 client-side asynchronous polling and callback methods defined by JAX-WS. For SCA service interfaces  
1790 defined using interface.java, the Java interface MUST NOT contain these methods. If these methods are  
1791 present, SCA Runtimes MUST NOT include them in the SCA reference interface as defined by the  
1792 Assembly specification. These methods are recognized as follows.

1793 For each method M in the interface, if another method P in the interface has

- 1794 a. a method name that is M's method name with the characters "Async" appended, and
- 1795 b. the same parameter signature as M, and
- 1796 c. a return type of Response<R> where R is the return type of M

1797 then P is a JAX-WS polling method that isn't part of the SCA interface contract.

1798 For each method M in the interface, if another method C in the interface has

- 1799 a. a method name that is M's method name with the characters "Async" appended, and
- 1800 b. a parameter signature that is M's parameter signature with an additional final parameter of type  
1801 AsyncHandler<R> where R is the return type of M, and
- 1802 c. a return type of Future<?>

1803 then C is a JAX-WS callback method that isn't part of the SCA interface contract.

1804 As an example, an interface may be defined in WSDL as follows:

```
1805 <!-- WSDL extract -->  
1806 <message name="getPrice">
```

```
1807 <part name="ticker" type="xsd:string"/>
1808 </message>
1809
1810 <message name="getPriceResponse">
1811 <part name="price" type="xsd:float"/>
1812 </message>
1813
1814 <portType name="StockQuote">
1815 <operation name="getPrice">
1816 <input message="tns:getPrice"/>
1817 <output message="tns:getPriceResponse"/>
1818 </operation>
1819 </portType>
```

1820

1821 The JAX-WS asynchronous mapping will produce the following Java interface:

```
1822 // asynchronous mapping
1823 @WebService
1824 public interface StockQuote {
1825     float getPrice(String ticker);
1826     Response<Float> getPriceAsync(String ticker);
1827     Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
1828 }
```

1829

1830 For SCA interface definition purposes, this is treated as equivalent to the following:

```
1831 // synchronous mapping
1832 @WebService
1833 public interface StockQuote {
1834     float getPrice(String ticker);
1835 }
```

1836

1837 SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above  
1838 example, if the client implementation uses the asynchronous form of the interface, the two  
1839 additional getPriceAsync() methods can be used for polling and callbacks as defined by the JAX-  
1840 WS specification.

---

1841 **10 Policy Annotations for Java**

1842 SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which  
1843 influence how implementations, services and references behave at runtime. The policy facilities  
1844 are described in [the SCA Policy Framework specification \[POLICY\]](#). In particular, the facilities  
1845 include Intents and Policy Sets, where intents express abstract, high-level policy requirements and  
1846 policy sets express low-level detailed concrete policies.

1847 Policy metadata can be added to SCA assemblies through the means of declarative statements  
1848 placed into Composite documents and into Component Type documents. These annotations are  
1849 completely independent of implementation code, allowing policy to be applied during the assembly  
1850 and deployment phases of application development.

1851 However, it can be useful and more natural to attach policy metadata directly to the code of  
1852 implementations. This is particularly important where the policies concerned are relied on by the  
1853 code itself. An example of this from the Security domain is where the implementation code  
1854 expects to run under a specific security Role and where any service operations invoked on the  
1855 implementation must be authorized to ensure that the client has the correct rights to use the  
1856 operations concerned. By annotating the code with appropriate policy metadata, the developer  
1857 can rest assured that this metadata is not lost or forgotten during the assembly and deployment  
1858 phases.

1859 The SCA Java Common Annotations specification provides a series of annotations which provide  
1860 the capability for the developer to attach policy information to Java implementation code. The  
1861 annotations concerned first provide general facilities for attaching SCA Intents and Policy Sets to  
1862 Java code. Secondly, there are further specific annotations that deal with particular policy intents  
1863 for certain policy domains such as Security.

1864 The SCA Java Common Annotations specification supports using [the Common Annotation for Java  
1865 Platform specification \(JSR-250\) \[JSR-250\]](#). An implication of adopting the common annotation  
1866 for Java platform specification is that the SCA Java specification support consistent annotation and  
1867 Java class inheritance relationships.

1868

## 1869 **10.1 General Intent Annotations**

1870 SCA provides the annotation **@Requires** for the attachment of any intent to a Java class, to a  
1871 Java interface or to elements within classes and interfaces such as methods and fields.

1872 The @Requires annotation can attach one or multiple intents in a single statement.

1873 Each intent is expressed as a string. Intents are XML QNames, which consist of a Namespace URI  
1874 followed by the name of the Intent. The precise form used follows the string representation used  
1875 by the `javax.xml.namespace.QName` class, which is as follows:

1876 `"{" + Namespace URI + "}" + intentname`

1877 Intents may be qualified, in which case the string consists of the base intent name, followed by a  
1878 ". ", followed by the name of the qualifier. There may also be multiple levels of qualification.

1879 This representation is quite verbose, so we expect that reusable String constants will be defined  
1880 for the namespace part of this string, as well as for each intent that is used by Java code. SCA  
1881 defines constants for intents such as the following:

```
1882 public static final String SCA_PREFIX="{http://docs.oasis-  
1883 open.org/ns/opencsa/sca/200712}";  
1884 public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";  
1885 public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
```

1886 Notice that, by convention, qualified intents include the qualifier as part of the name of the  
1887 constant, separated by an underscore. These intent constants are defined in the file that defines  
1888 an annotation for the intent (annotations for intents, and the formal definition of these constants,  
1889 are covered in a following section).

1890 Multiple intents (qualified or not) are expressed as separate strings within an array declaration.

1891 An example of the @Requires annotation with 2 qualified intents (from the Security domain)  
1892 follows:

1893

```
1894     @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

1895

1896 This attaches the intents "confidentiality.message" and "integrity.message".

1897 The following is an example of a reference requiring support for confidentiality:

```
1898     package org.oasisopen.sca.annotations;
```

1899

```
1900     import static org.oasisopen.sca.annotations.Confidentiality.*;
```

1901

```
1902     public class Foo {
```

```
1903         @Requires(CONFIDENTIALITY)
```

```
1904         @Reference
```

```
1905         public void setBar(Bar bar) {
```

```
1906             ...
```

```
1907         }
```

```
1908     }
```

1909 Users may also choose to only use constants for the namespace part of the QName, so that they  
1910 may add new intents without having to define new constants. In that case, this definition would  
1911 instead look like this:

```
1912     package org.oasisopen.sca.annotations;
```

1913

```
1914     import static org.oasisopen.sca.Constants.*;
```

1915

```
1916     public class Foo {
```

```
1917         @Requires(SCA_PREFIX+"confidentiality")
```

```
1918         @Reference
```

```
1919         public void setBar(Bar bar) {
```

```
1920             ...
```

```
1921         }
```

```
1922     }
```

1923

1924 The formal syntax for the @Requires annotation follows:

```
1925     @Requires( "qualifiedIntent" | {"qualifiedIntent" [, "qualifiedIntent"]}
```

```
1926     where
```

1927           qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2

1928

1929           The following shows the formal definition of the @Requires annotation:

1930

```
1931       package org.oasisopen.sca.annotations;
1932       import static java.lang.annotation.ElementType.TYPE;
1933       import static java.lang.annotation.ElementType.METHOD;
1934       import static java.lang.annotation.ElementType.FIELD;
1935       import static java.lang.annotation.ElementType.PARAMETER;
1936       import static java.lang.annotation.RetentionPolicy.RUNTIME;
1937       import java.lang.annotation.Retention;
1938       import java.lang.annotation.Target;
1939       import java.lang.annotation.Inherited;
```

1940

```
1941       @Inherited
1942       @Retention(RUNTIME)
1943       @Target({TYPE, METHOD, FIELD, PARAMETER})
```

1944

```
1945       public @interface Requires {
1946           String[] value() default "";
1947       }
```

1948           The SCA\_NS constant is defined in the Constants interface:

```
1949       package org.oasisopen.sca;
1950
1951       public interface Constants {
1952           String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";
1953           String SCA_PREFIX = "{"+SCA_NS+"}";
1954       }
```

1955

## 1956   10.2 Specific Intent Annotations

1957           In addition to the general intent annotation supplied by the @Requires annotation described  
1958           above, it is also possible to have Java annotations that correspond to specific policy intents. SCA  
1959           provides a number of these specific intent annotations and it is also possible to create new specific  
1960           intent annotations for any intent.

1961           The general form of these specific intent annotations is an annotation with a name derived from  
1962           the name of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an  
1963           attribute to the annotation in the form of a string or an array of strings.

1964           For example, the SCA confidentiality intent described in [the section on General Intent Annotations](#)  
1965           using the @Requires(CONFIDENTIALITY) intent can also be specified with the specific  
1966           @Confidentiality intent annotation. The specific intent annotation for the "integrity" security intent  
1967           is:

1968           @Integrity

1969           An example of a qualified specific intent for the "authentication" intent is:

1970           @Authentication( {"message", "transport"} )

1971           This annotation attaches the pair of qualified intents: "authentication.message" and  
1972           "authentication.transport" (the sca: namespace is assumed in this both of these cases –  
1973           "http://docs.oasis-open.org/ns/opencsa/sca/200712").

1974           The general form of specific intent annotations is:

1975           @<Intent>[(qualifiers)]

1976           where Intent is an NCName that denotes a particular type of intent.

1977           Intent ::= NCName

1978           qualifiers ::= "qualifier" | {"qualifier" [, "qualifier" ] }

1979           qualifier ::= NCName | NCName/qualifier

1980

## 1981   10.2.1 How to Create Specific Intent Annotations

1982           SCA identifies annotations that correspond to intents by providing an @Intent annotation which  
1983           must be used in the definition of an intent annotation.

1984           The @Intent annotation takes a single parameter, which (like the @Requires annotation) is the  
1985           String form of the QName of the intent. As part of the intent definition, it is good practice  
1986           (although not required) to also create String constants for the Namespace, the Intent and for  
1987           Qualified versions of the Intent (if defined). These String constants are then available for use with  
1988           the @Requires annotation and it should also be possible to use one or more of them as  
1989           parameters to the @Intent annotation.

1990           Alternatively, the QName of the intent may be specified using separate parameters for the  
1991           targetNamespace and the localPart for example:

1992           @Intent(targetNamespace=SCA\_NS, localPart="confidentiality").

1993           The definition of the @Intent annotation is the following:

1994

1995           package org.oasisopen.sca.annotations;

1996           import static java.lang.annotation.ElementType.ANNOTATION\_TYPE;

1997           import static java.lang.annotation.RetentionPolicy.RUNTIME;

1998           import java.lang.annotation.Retention;

1999           import java.lang.annotation.Target;

2000           import java.lang.annotation.Inherited;

2001

2002           @Retention(RUNTIME)

2003           @Target(ANNOTATION\_TYPE)

2004           public @interface Intent {

2005                 String value() default "";

2006                 String targetNamespace() default "";

2007                 String localPart() default "";

2008           }

2009 When an intent can be qualified, it is good practice for the first attribute of the annotation to be a  
2010 string (or an array of strings) which holds one or more qualifiers.

2011 In this case, the attribute's definition should be marked with the @Qualifier annotation. The  
2012 @Qualifier tells SCA that the value of the attribute should be treated as a qualifier for the intent  
2013 represented by the whole annotation. If more than one qualifier value is specified in an  
2014 annotation, it means that multiple qualified forms are required. For example:

```
2015 @Confidentiality( {"message", "transport" } )
```

2016 implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport"  
2017 are set for the element to which the confidentiality intent is attached.

2018 The following is the definition of the @Qualifier annotation.

2019

```
2020 package org.oasisopen.sca.annotations;
```

```
2021 import static java.lang.annotation.ElementType.METHOD;
```

```
2022 import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```
2023 import java.lang.annotation.Retention;
```

```
2024 import java.lang.annotation.Target;
```

```
2025 import java.lang.annotation.Inherited;
```

2026

```
2027 @Retention(RetentionPolicy.RUNTIME)
```

```
2028 @Target(ElementType.METHOD)
```

```
2029 public @interface Qualifier {
```

```
2030 }
```

2031

2032 Examples of the use of the @Intent and the @Qualifier annotations in the definition of specific  
2033 intent annotations are shown in [the section dealing with Security Interaction Policy](#).

2034

## 2035 10.3 Application of Intent Annotations

2036 The SCA Intent annotations can be applied to the following Java elements:

- 2037 • Java class
- 2038 • Java interface
- 2039 • Method
- 2040 • Field

2041 Where multiple intent annotations (general or specific) are applied to the same Java element, they  
2042 are additive in effect. An example of multiple policy annotations being used together follows:

```
2043 @Authentication
```

```
2044 @Requires( {CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE} )
```

2045 In this case, the effective intents are "authentication", "confidentiality.message" and  
2046 "integrity.message".

2047 If an annotation is specified at both the class/interface level and the method or field level, then  
2048 the method or field level annotation completely overrides the class level annotation of the same  
2049 type.

2050 The intent annotation can be applied either to classes or to class methods when adding annotated  
2051 policy on SCA services. Applying an intent to the setter method in a reference injection approach  
2052 allows intents to be defined at references.

### 2053 10.3.1 Inheritance And Annotation

2054 The inheritance rules for annotations are consistent with the common annotation specification, JSR  
2055 250.

2056 The following example shows the inheritance relations of intents on classes, operations, and super  
2057 classes.

```
2058
2059 package services.hello;
2060 import org.oasisopen.sca.annotations.Remotable;
2061 import org.oasisopen.sca.annotations.Integrity;
2062 import org.oasisopen.sca.annotations.Authentication;
2063
2064 @Integrity("transport")
2065 @Authentication
2066 public class HelloService {
2067     @Integrity
2068     @Authentication("message")
2069     public String hello(String message) {...}
2070
2071     @Integrity
2072     @Authentication("transport")
2073     public String helloThere() {...}
2074 }
2075
2076 package services.hello;
2077 import org.oasisopen.sca.annotations.Remotable;
2078 import org.oasisopen.sca.annotations.Confidentiality;
2079 import org.oasisopen.sca.annotations.Authentication;
2080
2081 @Confidentiality("message")
2082 public class HelloChildService extends HelloService {
2083     @Confidentiality("transport")
2084     public String hello(String message) {...}
2085     @Authentication
2086     String helloWorld() {...}
2087 }
```

2088 Example 2a. Usage example of annotated policy and inheritance.

2089

2090 The effective intent annotation on the helloWorld method is Integrity("transport"),  
2091 @Authentication, and @Confidentiality("message").

2092 The effective intent annotation on the hello method of the HelloChildService is  
 2093 @Integrity("transport"), @Authentication, and @Confidentiality("transport"),  
 2094 The effective intent annotation on the helloThere method of the HelloChildService is @Integrity  
 2095 and @Authentication("transport"), the same as in HelloService class.  
 2096 The effective intent annotation on the hello method of the HelloService is @Integrity and  
 2097 @Authentication("message")  
 2098  
 2099 The listing below contains the equivalent declarative security interaction policy of the HelloService  
 2100 and HelloChildService implementation corresponding to the Java interfaces and classes shown in  
 2101 Example 2a.

```

2102
2103 <?xml version="1.0" encoding="ASCII"?>
2104
2105 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2106           name="HelloServiceComposite" >
2107   <service name="HelloService" requires="integrity/transport
2108           authentication">
2109     ...
2110   </service>
2111   <service name="HelloChildService" requires="integrity/transport
2112           authentication confidentiality/message">
2113     ...
2114   </service>
2115   ...
2116
2117   <component name="HelloServiceComponent">*
2118     <implementation.java class="services.hello.HelloService"/>
2119     <operation name="hello" requires="integrity
2120           authentication/message"/>
2121     <operation name="helloThere"
2122   requires="integrity
2123           authentication/transport"/>
2124   </component>
2125   <component name="HelloChildServiceComponent">*
2126     <implementation.java
2127   class="services.hello.HelloChildService" />
2128     <operation name="hello"
2129   requires="confidentiality/transport"/>
2130     <operation name="helloThere" requires=" integrity/transport
2131           authentication"/>
2132     <operation name=helloWorld" requires="authentication"/>
2133   </component>
2134   ...
2135
2136 </composite>
  
```

2139 Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.

2140

## 2141 10.4 Relationship of Declarative And Annotated Intents

2142 Annotated intents on a Java class cannot be overridden by declarative intents either in a  
 2143 composite document which uses the class as an implementation or by statements in a component

2144 Type document associated with the class. This rule follows the general rule for intents that they  
2145 represent fundamental requirements of an implementation.

2146 An unqualified version of an intent expressed through an annotation in the Java class may be  
2147 qualified by a declarative intent in a using composite document.

2148

## 2149 10.5 Policy Set Annotations

2150 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies (for  
2151 example, a concrete policy is the specific encryption algorithm to use when encrypting messages  
2152 when using a specific communication protocol to link a reference to a service).

2153 Policy Sets can be applied directly to Java implementations using the **@PolicySets** annotation.  
2154 The @PolicySets annotation either takes the QName of a single policy set as a string or the name  
2155 of two or more policy sets as an array of strings:  
2156

```
2157     @PolicySets( "<policy set QName>" |  
2158               { "<policy set QName>" [, "<policy set QName>" ] })
```

2159

2160 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

2161 An example of the @PolicySets annotation:

2162

```
2163     @Reference(name="helloService", required=true)  
2164     @PolicySets({ MY_NS + "WS_Encryption_Policy",  
2165                MY_NS + "WS_Authentication_Policy" })  
2166     public setHelloService(HelloService service) {  
2167         . . .  
2168     }
```

2169 In this case, the Policy Sets WS\_Encryption\_Policy and WS\_Authentication\_Policy are applied, both  
2170 using the namespace defined for the constant MY\_NS.

2171 PolicySets must satisfy intents expressed for the implementation when both are present, according  
2172 to the rules defined in [the Policy Framework specification \[POLICY\]](#).

2173 The SCA Policy Set annotation can be applied to the following Java elements:

- 2174 • Java class
- 2175 • Java interface
- 2176 • Method
- 2177 • Field

2178

## 2179 10.6 Security Policy Annotations

2180 This section introduces annotations for SCA's security intents, as defined in [the SCA Policy  
2181 Framework specification \[POLICY\]](#).

2182

### 2183 10.6.1 Security Interaction Policy

2184 The following interaction policy Intents and qualifiers are defined for Security Policy, which apply  
2185 to the operation of services and references of an implementation:

- 2186 • @Integrity
- 2187 • @Confidentiality
- 2188 • @Authentication

2189 All three of these intents have the same pair of Qualifiers:

- 2190 • message
- 2191 • transport

2192 The following snippets shows the @Integrity, @Confidentiality and @Authentication annotations:

```

2193 package org.oasisopen.sca.annotations;
2194
2195 import java.lang.annotation.*;
2196 import static org.oasisopen.sca.Constants.SCA_NS;
2197
2198 @Inherited
2199 @Retention(RetentionPolicy.RUNTIME)
2200 @Target({ElementType.TYPE, ElementType.METHOD,
2201          ElementType.FIELD, ElementType.PARAMETER})
2202 @Intent(Integrity.INTEGRITY)
2203 public @interface Integrity {
2204     String INTEGRITY = SCA_NS+"integrity";
2205     String INTEGRITY_MESSAGE = INTEGRITY+".message";
2206     String INTEGRITY_TRANSPORT = INTEGRITY+".transport";
2207     @Qualifier
2208     String[] value() default "";
2209 }
2210
2211
2212 package org.oasisopen.sca.annotations;
2213
2214 import java.lang.annotation.*;
2215 import static org.oasisopen.sca.Constants.SCA_NS;
2216
2217 @Inherited
2218 @Retention(RetentionPolicy.RUNTIME)
2219 @Target({ElementType.TYPE, ElementType.METHOD,
2220          ElementType.FIELD, ElementType.PARAMETER})
2221 @Intent(Confidentiality.CONFIDENTIALITY)
2222 public @interface Confidentiality {
2223     String CONFIDENTIALITY = SCA_NS+"confidentiality";
2224     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY+".message";

```

```

2225     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY+".transport";
2226     @Qualifier
2227     String[] value() default "";
2228 }

```

2229  
2230

```
2231 package org.oasisopen.sca.annotations;
```

2232

```
2233 import java.lang.annotation.*;
```

```
2234 import static org.oasisopen.sca.Constants.SCA_NS;
```

2235

```
2236 @Inherited
```

```
2237 @Retention(RetentionPolicy.RUNTIME)
```

```
2238 @Target({ElementType.TYPE,ElementType.METHOD,
```

```
2239             ElementType.FIELD, ElementType.PARAMETER})
```

```
2240 @Intent(Authentication.AUTHENTICATION)
```

```
2241 public @interface Authentication {
```

```
2242     String AUTHENTICATION = SCA_NS+"authentication";
```

```
2243     String AUTHENTICATION_MESSAGE = AUTHENTICATION+".message";
```

```
2244     String AUTHENTICATION_TRANSPORT = AUTHENTICATION+".transport";
```

```
2245     @Qualifier
```

```
2246     String[] value() default "";
```

```
2247 }
```

2248

2249

2250 The following example shows an example of applying an intent to the setter method used to inject  
2251 a reference. Accessing the hello operation of the referenced HelloService requires both  
2252 "integrity.message" and "authentication.message" intents to be honored.

2253

```
2254 //Interface for HelloService
```

```
2255 public interface service.hello.HelloService {
```

```
2256     String hello(String helloMsg);
```

```
2257 }
```

2258

```
2259 // Interface for ClientService
```

```
2260 public interface service.client.ClientService {
```

```
2261     public void clientMethod();
```

```
2262 }
```

2263

```
2264 // Implementation class for ClientService
```

```
2265 package services.client;
```

```

2266
2267     import services.hello.HelloService;
2268
2269     import org.oasisopen.sca.annotations.*;
2270
2271     @Service(ClientService.class)
2272     public class ClientServiceImpl implements ClientService {
2273
2274
2275         private HelloService helloService;
2276
2277         @Reference(name="helloService", required=true)
2278         @Integrity("message")
2279         @Authentication("message")
2280         public void setHelloService(HelloService service) {
2281             helloService = service;
2282         }
2283
2284         public void clientMethod() {
2285             String result = helloService.hello("Hello World!");
2286             ...
2287         }
2288     }
2289

```

2290 Example 1. Usage of annotated intents on a reference.

2291

## 2292 10.6.2 Security Implementation Policy

2293 SCA defines a number of security policy annotations that apply as policies to implementations  
2294 themselves. These annotations mostly have to do with authorization and security identity. The  
2295 following authorization and security identity annotations (as defined in JSR 250) are supported:

- 2296 • RunAs
- 2297 Takes as a parameter a string which is the name of a Security role.  
2298 eg. @RunAs("Manager")
- 2300 • Code marked with this annotation will execute with the Security permissions of the  
2301 identified role.
- 2302 • RolesAllowed
- 2303 Takes as a parameter a single string or an array of strings which represent one or more  
2304 role names. When present, the implementation can only be accessed by principals whose  
2305 role corresponds to one of the role names listed in the @roles attribute. How role names  
2306 are mapped to security principals is implementation dependent (SCA does not define this).  
2307 eg. @RolesAllowed( {"Manager", "Employee"} )
- 2308
- 2309 • PermitAll
- 2310 No parameters. When present, grants access to all roles.
- 2311

- 2312 • DenyAll
- 2313
- 2314 No parameters. When present, denies access to all roles.
- 2315 • DeclareRoles
- 2316 Takes as a parameter a string or an array of strings which identify one or more role names
- 2317 that form the set of roles used by the implementation.
- 2318 eg. @DeclareRoles({"Manager", "Employee", "Customer"} )
- 2319 (all these are declared in the Java package javax.annotation.security)
- 2320 For a full explanation of these intents, see [the Policy Framework specification \[POLICY\]](#).

### 2321 10.6.2.1 Annotated Implementation Policy Example

2322 The following is an example showing annotated security implementation policy:

```
2323
2324 package services.account;
2325 @Remotable
2326 public interface AccountService {
2327     AccountReport getAccountReport(String customerID);
2328 }
```

2329

2330 The following is a full listing of the AccountServiceImpl class, showing the Service it implements,

2331 plus the service references it makes and the settable properties that it has, along with a set of

2332 implementation policy annotations:

```
2333
2334 package services.account;
2335 import java.util.List;
2336 import commonj.sdo.DataFactory;
2337 import org.oasisopen.sca.annotations.Property;
2338 import org.oasisopen.sca.annotations.Reference;
2339 import org.oasisopen.sca.annotations.RolesAllowed;
2340 import org.oasisopen.sca.annotations.RunAs;
2341 import org.oasisopen.sca.annotations.PermitAll;
2342 import services.accountdata.AccountDataService;
2343 import services.accountdata.CheckingAccount;
2344 import services.accountdata.SavingsAccount;
2345 import services.accountdata.StockAccount;
2346 import services.stockquote.StockQuoteService;
2347 @RolesAllowed("customers")
2348 @RunAs("accountants" )
2349 public class AccountServiceImpl implements AccountService {
2350
2351     @Property
2352     protected String currency = "USD";
2353
2354     @Reference
2355     protected AccountDataService accountDataService;
```

```

2356     @Reference
2357     protected StockQuoteService stockQuoteService;
2358
2359     @RolesAllowed({"customers", "accountants"})
2360     public AccountReport getAccountReport(String customerID) {
2361
2362         DataFactory dataFactory = DataFactory.INSTANCE;
2363         AccountReport accountReport =
2364             (AccountReport)dataFactory.create(AccountReport.class);
2365         List accountSummaries = accountReport.getAccountSummaries();
2366
2367         CheckingAccount checkingAccount =
2368             accountDataService.getCheckingAccount(customerID);
2369         AccountSummary checkingAccountSummary =
2370             (AccountSummary)dataFactory.create(AccountSummary.class);
2371
2372         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber()
2373         );
2374         checkingAccountSummary.setAccountType("checking");
2375         checkingAccountSummary.setBalance(fromUSDollarToCurrency
2376             (checkingAccount.getBalance()));
2377         accountSummaries.add(checkingAccountSummary);
2378
2379         SavingsAccount savingsAccount =
2380             accountDataService.getSavingsAccount(customerID);
2381         AccountSummary savingsAccountSummary =
2382             (AccountSummary)dataFactory.create(AccountSummary.class);
2383
2384         savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
2385         savingsAccountSummary.setAccountType("savings");
2386         savingsAccountSummary.setBalance(fromUSDollarToCurrency
2387             (savingsAccount.getBalance()));
2388         accountSummaries.add(savingsAccountSummary);
2389
2390         StockAccount stockAccount =
2391         accountDataService.getStockAccount(customerID);
2392         AccountSummary stockAccountSummary =
2393             (AccountSummary)dataFactory.create(AccountSummary.class);
2394         stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
2395         stockAccountSummary.setAccountType("stock");
2396         float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*
2397             stockAccount.getQuantity();
2398         stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
2399         accountSummaries.add(stockAccountSummary);

```

```

2400
2401     return accountReport;
2402 }
2403
2404 @PermitAll
2405 public float fromUSDollarToCurrency(float value) {
2406
2407     if (currency.equals("USD")) return value; else
2408     if (currency.equals("EURO")) return value * 0.8f; else
2409     return 0.0f;
2410 }
2411 }

```

2412 Example 3. Usage of annotated security implementation policy for the java language.

2413 In this example, the implementation class as a whole is marked:

- 2414 • @RolesAllowed("customers") - indicating that customers have access to the
- 2415 implementation as a whole
- 2416 • @RunAs("accountants" ) - indicating that the code in the implementation runs with the
- 2417 permissions of accountants

2418 The getAccountReport(..) method is marked with @RolesAllowed({"customers", "accountants"}),  
2419 which indicates that this method can be called by both customers and accountants.

2420 The fromUSDollarToCurrency() method is marked with @PermitAll, which means that this method  
2421 can be called by any role.

2422

---

## 2423 A. XML Schema: sca-interface-java.xsd

```
2424 <?xml version="1.0" encoding="UTF-8"?>
2425 <!-- (c) Copyright SCA Collaboration 2006 -->
2426 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2427         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2428         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2429         elementFormDefault="qualified">
2430
2431     <include schemaLocation="sca-core.xsd"/>
2432
2433     <element name="interface.java" type="sca:JavaInterface"
2434             substitutionGroup="sca:interface"/>
2435     <complexType name="JavaInterface">
2436         <complexContent>
2437             <extension base="sca:Interface">
2438                 <sequence>
2439                     <any namespace="##other" processContents="lax"
2440 minOccurs="0"                                maxOccurs="unbounded"/>
2441                 </sequence>
2442                 <attribute name="interface" type="NCName" use="required"/>
2443                 <attribute name="callbackInterface" type="NCName"
2444 use="optional"/>
2445                 <anyAttribute namespace="##any" processContents="lax"/>
2446             </extension>
2447         </complexContent>
2448     </complexType>
2449 </schema>
2450
```

2451

---

## B. Conformance Items

2452 This section contains a list of conformance items for the SCA Java Common Annotations and APIs  
2453 specification.

2454

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.

2455

2456

---

## C. Acknowledgements

2457 The following individuals have participated in the creation of this specification and are gratefully  
2458 acknowledged:

2459 **Participants:**

2460 [Participant Name, Affiliation | Individual Member]

2461 [Participant Name, Affiliation | Individual Member]

2462

---

## D. Non-Normative Text

2464

## E. Revision History

2465 [optional; should not be included in OASIS Standards]

2466

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.
--	--	--	-----------------------

2467

2468