



# Service Component Architecture Java Common Annotations and APIs Specification Version 1.1 + Issue 25

Committee Draft 01

03 October 2008

**Specification URIs:**

**This Version:**

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

**Previous Version:**

**Latest Version:**

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

**Latest Approved Version:**

**Technical Committee:**

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

**Chair(s):**

Simon Nash,	IBM
Michael Rowley,	BEA Systems
Mark 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	Request scope.....	11
2.2.3	Composite scope.....	11
2.2.4	Conversation scope.....	11
3	Interface.....	12
3.1	Java interface element ("interface.java").....	12
3.2	@Remotable.....	12
3.3	@Conversational.....	12
4	Client API.....	13
4.1	Accessing Services from an SCA Component.....	13
4.1.1	Using the Component Context API.....	13
4.2	Accessing Services from non-SCA component implementations.....	13
4.2.1	ComponentContext.....	13
5	Error Handling.....	14
6	Asynchronous and Conversational Programming.....	15
6.1	@OneWay.....	15
6.2	Conversational Services.....	15
6.2.1	ConversationAttributes.....	15
6.2.2	@EndsConversation.....	16
6.3	Passing Conversational Services as Parameters.....	16
6.4	Conversational Client.....	16
6.5	Conversation Lifetime Summary.....	17
6.6	Conversation ID.....	18
6.6.1	Application Specified Conversation IDs.....	18
6.6.2	Accessing Conversation IDs from Clients.....	18
6.7	Callbacks.....	18
6.7.1	Stateful Callbacks.....	18
6.7.2	Stateless Callbacks.....	<a href="#">2220</a>
6.7.3	Implementing Multiple Bidirectional Interfaces.....	<a href="#">2224</a>
6.7.4	Accessing Callbacks.....	<a href="#">2324</a>
6.7.5	Customizing the Callback.....	<a href="#">2422</a>

6.7.6	Customizing the Callback Identity .....	<a href="#">2422</a>
6.7.7	Bindings for Conversations and Callbacks.....	<a href="#">2423</a>
7	Java API .....	<a href="#">2624</a>
7.1	Component Context.....	<a href="#">2624</a>
7.2	Request Context .....	<a href="#">2725</a>
7.3	CallableReference .....	<a href="#">2826</a>
7.4	ServiceReference .....	<a href="#">2826</a>
7.5	Conversation.....	<a href="#">2927</a>
7.6	ServiceRuntimeException.....	<a href="#">2927</a>
7.7	NoRegisteredCallbackException .....	<a href="#">3028</a>
7.8	ServiceUnavailableException .....	<a href="#">3028</a>
7.9	InvalidServiceException.....	<a href="#">3028</a>
7.10	ConversationEndedException .....	<a href="#">3028</a>
8	Java Annotations .....	<a href="#">3230</a>
8.1	@AllowsPassByReference.....	<a href="#">3230</a>
8.2	@Callback .....	<a href="#">3230</a>
8.3	@ComponentName .....	<a href="#">3432</a>
8.4	@Constructor.....	<a href="#">3432</a>
8.5	@Context .....	<a href="#">3533</a>
8.6	@Conversational .....	<a href="#">3634</a>
8.7	@ConversationAttributes.....	<a href="#">3634</a>
8.8	@ConversationID .....	<a href="#">3735</a>
8.9	@Destroy .....	<a href="#">3836</a>
8.10	@EagerInit.....	<a href="#">3836</a>
8.11	@EndsConversation.....	<a href="#">3937</a>
8.12	@Init.....	<a href="#">3937</a>
8.13	@OneWay .....	<a href="#">4038</a>
8.14	@Property.....	<a href="#">4139</a>
8.15	@Reference.....	<a href="#">4240</a>
8.15.1	Reinjection.....	<a href="#">4442</a>
8.16	@Remotable.....	<a href="#">4644</a>
8.17	@Scope .....	<a href="#">4745</a>
8.18	@Service .....	<a href="#">4846</a>
9	WSDL to Java and Java to WSDL .....	<a href="#">5048</a>
9.1	JAX-WS Client Asynchronous API for a Synchronous Service.....	<a href="#">5048</a>
10	Policy Annotations for Java .....	<a href="#">5250</a>
10.1	General Intent Annotations .....	<a href="#">5250</a>
10.2	Specific Intent Annotations .....	<a href="#">5452</a>
10.2.1	How to Create Specific Intent Annotations.....	<a href="#">5553</a>
10.3	Application of Intent Annotations .....	<a href="#">5654</a>
10.3.1	Inheritance And Annotation.....	<a href="#">5755</a>
10.4	Relationship of Declarative And Annotated Intents .....	<a href="#">5856</a>
10.5	Policy Set Annotations .....	<a href="#">5957</a>
10.6	Security Policy Annotations .....	<a href="#">5957</a>
10.6.1	Security Interaction Policy.....	<a href="#">5957</a>

10.6.2 Security Implementation Policy .....	<del>6260</del>
A. Acknowledgements .....	<del>6664</del>
B. Non-Normative Text .....	<del>6765</del>
C. Revision History.....	<del>6866</del>

---

# 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 [1]. It specifies a set of APIs and annotations that may be used by Java-based SCA specifications.

Specifically, this specification covers:

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

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

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

This document defines implementation metadata using the annotation capability from Java™ 2 Standard Edition (J2SE) 5. However, SCA also allows service clients and implementations to be written using J2SE 1.4. All metadata that is represented by annotations can also be expressed using a component type side file, as defined in the SCA Assembly Specification [1].

## 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, *Key words for use in RFCs to Indicate Requirement Levels*, <http://www.ietf.org/rfc/rfc2119.txt>, IETF RFC 2119, March 1997.

TBD TBD

[1] SCA Assembly Specification

<http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf>

[2] SDO 2.1 Specification

<http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf>

[3] JAXB Specification

<http://www.jcp.org/en/jsr/detail?id=31>

41 [4] WSDL Specification  
42 WSDL 1.1: <http://www.w3.org/TR/wsdl>  
43 WSDL 2.0: <http://www.w3.org/TR/wsdl20/>  
44 [5] SCA Policy Framework  
45 [http://www.osoa.org/download/attachments/35/SCA\\_Policy\\_Framework\\_V100.pdf](http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf)  
46 [6] Common Annotation for Java Platform specification (JSR-250)  
47 <http://www.jcp.org/en/jsr/detail?id=250>  
48 [7] JAX-WS Specification (JSR-224)  
49 <http://www.jcp.org/en/jsr/detail?id=224>  
50

### 51 **1.3 Non-Normative References**

52 **TBD** **TBD**



---

## 53 2 Implementation Metadata

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

### 56 2.1 Service Metadata

#### 57 2.1.1 @Service

58

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

- 61 • As a Java interface
- 62 • As a Java class
- 63 • As a Java interface generated from a Web Services Description Language [4] (WSDL)  
64 portType (Java interfaces generated from a WSDL portType are always **remotable**)

#### 65 2.1.2 Java Semantics of a Remotable Service

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

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

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

#### 77 2.1.3 Java Semantics of a Local Service

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

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

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

83

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

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

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

#### 94 2.1.4 @Reference

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

#### 98 2.1.5 @Property

99 Implementations can be configured with data values through the use of properties, as defined in  
100 the SCA Assembly specification [1]. The **@Property** annotation is used to define an SCA property .

### 101 2.2 Implementation Scopes: @Scope, @Init, @Destroy

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

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

108 This document defines four scopes:

- 109 • STATELESS
- 110 • REQUEST
- 111 • CONVERSATION
- 112 • COMPOSITE

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

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

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

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

121 Note that only no-argument methods may be annotated as lifecycle methods.

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

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

134

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

### 137 2.2.1 Stateless scope

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

### 140 2.2.2 Request scope

141 The lifecycle of request scope extends from the point a request on a remotable interface enters  
142 the SCA runtime and a thread processes that request until the thread completes synchronously  
143 processing the request. During that time, all service requests are delegated to the same  
144 implementation instance of a request-scoped component.

145 There are times when a local request scoped service is called without there being a remotable  
146 service earlier in the call stack, such as when a local service is called from a non-SCA entity. In  
147 these cases, a remote request is always considered to be present, but the lifetime of the request is  
148 implementation dependent. For example, a timer event could be treated as a remote request.

### 149 2.2.3 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 2.2.4 Conversation scope

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

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

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

---

## 172 3 Interface

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

### 174 3.1 Java interface element ("interface.java")

175 The following snippet shows the schema for the Java interface element.

176

```
177 <interface.java interface="NCName" ... />
```

178

179 The interface.java element has the following attributes:

- 180 • **interface** – the fully qualified name of the Java interface

181

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

183

```
184 <interface.java interface="services.stockquote.StockQuoteService"/>
```

185

186 Here, the Java interface is defined in the Java class file  
187 ./services/stockquote/StockQuoteService.class, where the root directory is defined by the  
188 contribution in which the interface exists.

189 For the Java interface type system, **arguments and return values** of the service methods are  
190 described using Java classes or simple Java types. [Service Data Objects \[2\]](#) are the preferred form  
191 of Java class because of their integration with XML technologies.

192

### 193 3.2 @Remotable

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

### 198 3.3 @Conversational

199 Java service interfaces may be annotated to specify whether their contract is conversational as  
200 described in [the Assembly Specification \[1\]](#) by using the **@Conversational** annotation. A  
201 conversational service indicates that requests to the service are correlated in some way.

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

---

## 203 4 Client API

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

### 206 4.1 Accessing Services from an SCA Component

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

#### 211 4.1.1 Using the Component Context API

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

### 215 4.2 Accessing Services from non-SCA component implementations

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

#### 218 4.2.1 ComponentContext

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

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

```
223  
224 ComponentContext context = // obtained through host environment-specific means  
225 HelloService helloService =  
226     context.getService(HelloService.class, "HelloService");  
227 String result = helloService.hello("Hello World!");
```

---

## 228 **5 Error Handling**

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

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

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

---

## 235 6 Asynchronous and Conversational Programming

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

- 243 • support for non-blocking method calls
- 244 • conversational services
- 245 • callbacks

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

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

### 251 6.1 @OneWay

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

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

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

### 262 6.2 Conversational Services

263 A service may be declared as conversational by marking its Java interface with an  
264 **@Conversational** annotation. If a service interface is not marked with **@Conversational**, it is  
265 stateless.

#### 266 6.2.1 ConversationAttributes

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

269 An example of **@ConversationAttributes** is shown below:

```
270 package com.bigbank;  
271 import org.osoa.sca.annotations.ConversationAttributes;  
272  
273 @ConversationAttributes(maxAge="30 days");  
274 public class LoanServiceImpl implements LoanService {  
275  
276 }
```

## 277 6.2.2 @EndsConversation

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

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

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

## 291 6.3 Passing Conversational Services as Parameters

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

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

```
297     interface ComponentContext{  
298         ...  
299         <B> ServiceReference<B> createSelfReference(Class  
300             businessInterface);  
301         <B> ServiceReference<B> createSelfReference(Class  
302             businessInterface, String serviceName);  
303     }
```

304  
305 The second variant, which takes an additional **serviceName** parameter, must be used if the  
306 component implements multiple services.

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

## 310 6.4 Conversational Client

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

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

```
318  
319     @Reference  
320     LoanService loanService;  
321     // Known to be conversational because the interface is marked as  
322     // conversational
```



```

323     public void applyForMortgage(Customer customer, HouseInfo houseInfo,
324                                 int term)
325     {
326         LoanApplication loanApp;
327         loanApp = createApplication(customer, houseInfo);
328         loanService.apply(loanApp);
329         loanService.lockCurrentRate(term);
330     }
331
332     public boolean isApproved() {
333         return loanService.getLoanStatus().equals("approved");
334     }
335     public LoanApplication createApplication(Customer customer,
336                                             HouseInfo houseInfo) {
337         return ...;
338     }

```

## 339 6.5 Conversation Lifetime Summary

### 340 **Starting conversations**

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

- 342 • A @Reference to a conversational service is injected
- 343 • A call is made to CompositeContext.getServiceReference and then a method of the service
- 344 is called.

### 346 **Continuing conversations**

347 The client can continue an existing conversation, by:

- 348 • Holding the service reference that was created when the conversation started
- 349 • Getting the service reference object passed as a parameter from another service, even
- 350 remotely
- 351 • Loading a service reference that had been written to some form of persistent storage

### 353 **Ending conversations**

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

- 355 • A service operation that has been annotated @EndsConversation has been called
- 356 • The server calls an @EndsConversation method on the @Callback reference
- 357 • The server's conversation lifetime timeout occurs
- 358 • The client calls Conversation.end()
- 359 • Any non-business exception is thrown by a conversational operation

360  
361 If a method is invoked on a service reference after an @EndsConversation method has been called  
362 then a new conversation will automatically be started. If  
363 ServiceReference.getConversationID() is called after the @EndsConversation method is called,  
364 but before the next conversation has been started, it returns null.

365 If a service reference is used after the service provider's conversation timeout has caused the  
366 conversation to be ended, then ConversationEndedException is thrown. In order to use that  
367 service reference for a new conversation, its endConversation () method must be called.  
368

## 369 6.6 Conversation ID

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

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

### 376 6.6.1 Application Specified Conversation IDs

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

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

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

### 385 6.6.2 Accessing Conversation IDs from Clients

386 Whether the conversation ID is chosen by the client or is generated by the system, the client may  
387 access the conversation ID by calling `getConversationID()` on the current conversation  
388 object.

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

## 392 6.7 Callbacks

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

- 397 • an interface for the provided service
- 398 • a callback interface that must be provided by the client

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

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

### 406 6.7.1 Stateless Callbacks

407 A **stateless** callback does not depend on the execution context of the callback method having  
408 access to the execution state of the method that originally invoked the bidirectional service. Any  
409 information needed by the callback method is either passed in the callback's parameters or is

410 obtained using data from these parameters. For example, information needed by the callback  
411 could be obtained from a database record that was retrieved using a key passed as a callback  
412 parameter.

413 A callback is stateless if its implementation has STATELESS or COMPOSITE scope. For a stateless-  
414 scoped implementation, the callback is dispatched using a newly initialized instance that doesn't  
415 share any state with the instance that made the original bidirectional service invocation. For a  
416 composite-scoped implementation, a single copy of the component's state is shared by all its  
417 methods including callbacks, so the callback's execution context might or might not contain the  
418 same execution state as the method that invoked the bidirectional service.

419 The following example interfaces show a bidirectional interface with a stateless callback.

```
420 package somepackage;  
421 import org.oesa.sca.annotations.Callback;  
422 import org.oesa.sca.annotations.Remotable;  
423 @Remotable  
424 @Callback(OrderServiceCallback.class)  
425 public interface OrderService {  
426     void queryStatus(String id);  
427 }  
428  
429 @Remotable  
430 public interface OrderServiceCallback {  
431     void updateStatus(String id, String status);  
432 }  
433
```

434 In this example, the queryStatus operation requests an update on the current status of an  
435 outstanding order which is identified by an order number. The order status is returned using the  
436 updateStatus callback operation, which includes the order number as well as the current status of  
437 the order.

438 The following code snippet illustrates a possible implementation of the example service, using the  
439 @Callback annotation to request that a callback proxy be injected. In this example, the service  
440 makes between zero and three callbacks with information about the status of the order.

```
441  
442 @Callback  
443 protected OrderServiceCallback callback;  
444  
445 public void queryStatus(String id) {  
446     if (isInvoiced(id)) {  
447         callback.updateStatus(id, "invoiced");  
448     }  
449     if (isDispatched(id)) {  
450         callback.updateStatus(id, "dispatched");  
451     }  
452     if (isPaid(id)) {  
453         callback.updateStatus(id, "paid");  
454     }  
455 }  
456
```

457 The code snippet below is taken from the client of this example service. The client's service  
458 implementation class implements the methods of the OrderServiceCallback interface as well as  
459 those of its service interface.

460

```

461 public class ClientImpl implements ClientService, OrderServiceCallback {
462
463     private OrderService myService;
464
465     @Reference
466     public void setMyService(OrderService service) {
467         myService = service;
468     }
469
470     public void aClientMethod() {
471         ...
472         myService.queryStatus(id);
473     }
474
475     public void updateStatus(String id, String status) {
476         // code to process the status update
477     }
478 }
479

```

480 Any correlation that the client needs to perform between service invocations and resulting  
481 callbacks is handled by business logic in the service and client implementations, using data passed  
482 as parameters of service and callback method invocations. If a client needs to store any  
483 persistent state to correlate service calls with subsequent callbacks, it is the responsibility of such  
484 a client to perform any persistent state management itself.

## 485 6.7.16.7.2 Stateful Callbacks

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

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

```

489 package somepackage;
490 import org.osoa.sca.annotations.Callback;
491 import org.osoa.sca.annotations.Conversational;
492 import org.osoa.sca.annotations.Remotable;
493 @Remotable
494 @Conversational
495 @Callback(MyServiceCallback.class)
496 public interface MyService {
497
498     void someMethod(String arg);
499 }
500
501 @Remotable
502 @Conversational
503 public interface MyServiceCallback {
504
505     void receiveResult(String result);
506 }
507

```

508 An implementation of the service in this example could use the @Callback annotation to request  
509 that a stateful callback be injected. The following is a fragment of an implementation of the

510 example service. In this example, the request is passed on to some other component, so that the  
511 example service acts essentially as an intermediary. If the example service is conversation  
512 scoped, the callback will still be available when the backend service sends back its asynchronous  
513 response.

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

```
518
519 @Callback
520 protected MyServiceCallback callback;
521
522 @Reference
523 protected MyService backendService;
524
525 public void someMethod(String arg) {
526     backendService.someMethod(arg);
527 }
528
529 public void receiveResult(String result) {
530     callback.receiveResult(result);
531 }
```

532

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

537

```
538
539 private MyService myService;
540
541 @Reference
542 public void setMyService(MyService service) {
543     myService = service;
544 }
545
546 public void aClientMethod() {
547     ...
548     myService.someMethod(arg);
549 }
550
551 public void receiveResult(String result) {
552     // code to process the result
553 }
```

554

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

563

## 6.7.2 Stateless Callbacks

564

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

571

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

572

573

574

```
private ServiceReference<MyService> myService;

@Reference
public void setMyService(ServiceReference<MyService> service) {
    myService = service;
}

public void aClientMethod() {
    String someKey = "1234";
    ...

    myService.setCallbackID(someKey);
    myService.getService().someMethod(arg);
}

public void receiveResult(String result) {
    Object key = myService.getCallbackID();
    // Lookup any relevant state based on "key"
    // code to process the result
}
```

575

576

577

578

579

580

581

582

583

584

585

586

587

588

589

590

591

592

593

594

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

597

598

599

```
@Callback
protected MyServiceCallback callback;
```

600

601

602

603

604

605

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

606

## 6.7.3 Implementing Multiple Bidirectional Interfaces

607

608

609

610

611

612

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

613

614

```
@Callback
```

```
615     protected MyService1Callback callback1;
616
617     @Callback
618     protected MyService2Callback callback2;
```

619

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

## 622 6.7.4 Accessing Callbacks

623 In addition to injecting a reference to a callback service, it is also possible to obtain a reference to  
624 a Callback instance by annotating a field or method [of type CallableReference](#) with the  
625 **@Callback** annotation.

626  
627 A reference implementing the callback service interface may be obtained using  
628 `CallableReference.getService()`.

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

```
630
631     @Callback
632     protected CallableReference<MyCallback> callback;
633
634     public void someMethod() {
635
636         MyCallback myCallback = callback.getCallback();    ...
637
638         myCallback.receiveResult(theResult);
639     }
640
```

641 [Because CallableReference objects are serializable, they can be stored persistently and retrieved](#)  
642 [at a later time to make a callback invocation after the associated service request has completed.](#)  
643 [CallableReference objects can also be passed as parameters on service invocations, enabling the](#)  
644 [responsibility for making the callback to be delegated to another service.](#)

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

```
647
648     public void someMethod() {
649
650         MyCallback myCallback =
651             ComponentContext.getRequestContext().getCallback();
652
653         ...
654
655         myCallback.receiveResult(theResult);
656     }
657
```

658 [This is necessary if the service implementation has COMPOSITE scope, because callback injection](#)  
659 [is not performed for composite-scoped implementations.](#)

660 [On the client side, the service that implements the callback can access the callback ID \(i.e.,](#)  
661 [reference parameters\) that was returned with the callback operation by accessing the request](#)  
662 [context, as follows:-](#)

663

```
664     @Context
```

```

665     protected RequestContext requestContext;
666     ---
667     void receiveResult(Object theResult) {
668     ---
669         Object refParams =
670     requestContext.getServiceReference().getCallbackID();
671         -----
672     }

```

674 On the client side, the object returned by the `getServiceReference()` method represents the  
675 service reference that was used to send the original request. The object returned by  
676 `getCallbackID()` represents the identity associated with the callback, which may be a single  
677 String or may be an object (as described below in "Customizing the Callback Identity").

## 678 6.7.5 Customizing the Callback

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

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

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

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

## 695 6.7.6 Customizing the Callback Identity

696 The identity that is used to identify a callback request is initially generated by the system.  
697 However, it is possible to provide an application-specified identity to identify the callback by calling  
698 the **`ServiceReference.setCallbackID()`** method. This can be used both for stateful and for  
699 stateless callbacks. The identity is sent to the service provider, and the binding must guarantee  
700 that the service provider will send the ID back when any callback method is invoked.

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

## 705 6.7.7 6.7.5 Bindings for Conversations and Callbacks

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



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

---

## 713 7 Java API

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

### 715 7.1 Component Context

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

```
717
718 package org.osoa.sca;
719
720 public interface ComponentContext {
721     String getURI();
722
723     <B> B getService(Class<B> businessInterface, String referenceName);
724
725     <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
726                                             String referenceName);
727
728     <B> Collection<B> getServices(Class<B> businessInterface,
729                               String referenceName);
730
731     <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
732                                                         businessInterface, String referenceName);
733
734     <B> ServiceReference<B> createSelfReference(Class<B>
735                                               businessInterface);
736
737     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
738                                               String serviceName);
739
740     <B> B getProperty(Class<B> type, String propertyName);
741
742     <B, R extends CallableReference<B>> R cast(B target)
743         throws IllegalArgumentException;
744
745     RequestContext getRequestContext();
746
747     <B> ServiceReference<B> cast(B target) throws IllegalArgumentException;
748 }
```

- 749
- 750 • **getURI()** - returns the absolute URI of the component within the SCA domain
  - 751 • **getService(Class<B> businessInterface, String referenceName)** – Returns a proxy for  
752 the reference defined by the current component. The `getService()` method takes as its  
753 input arguments the Java type used to represent the target service on the client and the  
754 name of the service reference. It returns an object providing access to the service. The  
755 returned object implements the Java interface the service is typed with. This method  
756 MUST throw an `IllegalArgumentException` if the reference has multiplicity greater than  
757 one.
  - 758 • **getServiceReference(Class<B> businessInterface, String referenceName)** – Returns a  
759 `ServiceReference` defined by the current component. This method MUST throw an  
760 `IllegalArgumentException` if the reference has multiplicity greater than one.

- 761 • **getServices(Class<B> businessInterface, String referenceName)** – Returns a list of  
762 typed service proxies for a business interface type and a reference name.
- 763 • **getServiceReferences(Class<B> businessInterface, String referenceName)** –Returns a  
764 list typed service references for a business interface type and a reference name.
- 765 • **createSelfReference(Class<B> businessInterface)** – Returns a ServiceReference that can  
766 be used to invoke this component over the designated service.
- 767 • **createSelfReference(Class<B> businessInterface, String serviceName)** – Returns a  
768 ServiceReference that can be used to invoke this component over the designated service.  
769 Service name explicitly declares the service name to invoke
- 770 • **getProperty (Class<B> type, String propertyName)** - Returns the value of an SCA  
771 property defined by this component.
- 772 • **getRequestContext()** - Returns the context for the current SCA service request, or null if  
773 there is no current request or if the context is unavailable. This method MUST return non-  
774 null when invoked during the execution of a Java business method for a service operation  
775 or callback operation, on the same thread that the SCA runtime provided, and MUST  
776 return null in all other cases.
- 777 • **cast(B target)** - Casts a type-safe reference to a CallableReference

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

781  
782 The following shows an example of component context usage in a Java class using the @Context  
783 annotation.

```
784 private ComponentContext componentContext;
785
786 @Context
787 public void setContext(ComponentContext context) {
788     componentContext = context;
789 }
790
791 public void doSomething() {
792     HelloWorld service =
793     componentContext.getService(HelloWorld.class, "HelloWorldComponent");
794     service.hello("hello");
795 }
796
```

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

## 800 7.2 Request Context

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

```
802
803 package org.oesoa.sca;
804
805 import javax.security.auth.Subject;
806
807 public interface RequestContext {
808
809     Subject getSecuritySubject();
810
```

```

811     String getServiceName();
812     <CB> CallableReference<CB> getCallbackReference();
813     <CB> CB getCallback();
814     <B> CallableReference<B> getServiceReference();
815
816 }
817

```

818 The RequestContext interface has the following methods:

- 819 • **getSecuritySubject()** – Returns the JAAS Subject of the current request
- 820 • **getServiceName()** – Returns the name of the service on the Java implementation the  
821 request came in on
- 822 • **getCallbackReference()** – Returns a callable reference to the callback as specified by the  
823 caller
- 824 • **getCallback()** – Returns a proxy for the callback as specified by the caller
- 825 • **getServiceReference()** – When invoked during the execution of a service operation, this  
826 method MUST return a CallableReference that represents the service that was invoked.  
827 When invoked during the execution of a callback operation, this method MUST return a  
828 CallableReference that represents the callback that was invoked.

## 829 7.3 CallableReference

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

```

831
832 package org.osoa.sca;
833
834 public interface CallableReference<B> extends java.io.Serializable {
835
836     B getService();
837     Class<B> getBusinessInterface();
838     boolean isConversational();
839     Conversation getConversation();
840     Object getCallbackID();
841 }
842

```

843 The CallableReference interface has the following methods:

- 844
- 845 • **getService()** - Returns a type-safe reference to the target of this reference. The instance  
846 returned is guaranteed to implement the business interface for this reference. The value  
847 returned is a proxy to the target that implements the business interface associated with this  
848 reference.
- 849 • **getBusinessInterface()** – Returns the Java class for the business interface associated with  
850 this reference.
- 851 • **isConversational()** – Returns true if this reference is conversational.
- 852 • **getConversation()** – Returns the conversation associated with this reference. Returns null if  
853 no conversation is currently active.
- 854 • ~~**getCallbackID()** – Returns the callback ID.~~

## 855 7.4 ServiceReference

856

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

860 The following Java code defines the ServiceReference interface:

```
861  
862 package org.osoa.sca;  
863  
864 public interface ServiceReference<B> extends CallableReference<B> {  
865     Object getConversationID();  
866     void setConversationID(Object conversationId) throws  
867         IllegalStateException;  
868     void setCallbackID(Object callbackID);  
869     Object getCallback();  
870     void setCallback(Object callback);  
871 }  
872
```

873

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

875

- 876 • **getConversationID()** - Returns the id supplied by the user that will be associated with  
877 future conversations initiated through this reference, or null if no ID has been set by the  
878 user.
- 879 • **setConversationID(Object conversationId)** – Set the ID, supplied by the user, to associate  
880 with any future conversation started through this reference. If the value supplied is null then  
881 the id will be generated by the implementation. Throws an IllegalStateException if a  
882 conversation is currently associated with this reference.
- 883 • ~~setCallbackID(Object callbackID) – Sets the callback ID.~~
- 884 • ~~getCallback() – Returns the callback object.~~
- 885 • ~~setCallback(Object callback) – Sets the callback object.~~

## 886 7.5 Conversation

887 The following snippet defines Conversation:

888

```
889 package org.osoa.sca;  
890  
891 public interface Conversation {  
892     Object getConversationID();  
893     void end();  
894 }
```

895

896 The Conversation interface has the following methods:

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

## 901 7.6 ServiceRuntimeException

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

903  
904  
905  
906  
907  
908  
909  
910

```
package org.osoa.sca;  
  
public class ServiceRuntimeException extends RuntimeException {  
    ...  
}
```

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

## 911 **7.7 NoRegisteredCallbackException**

912 The following snippet shows the ~~NoRegisteredCallbackException~~.

913  
914  
915  
916  
917  
918  
919  
920  
921  
922

```
package org.osoa.sca;  
  
public class NoRegisteredCallbackException extends  
    ServiceRuntimeException {  
    ...  
}
```

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

## 923 **7.87.7 ServiceUnavailableException**

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

925  
926  
927  
928  
929  
930  
931

```
package org.osoa.sca;  
  
public class ServiceUnavailableException extends ServiceRuntimeException {  
    ...  
}
```

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

## 936 **7.97.8 InvalidServiceException**

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

938  
939  
940  
941  
942  
943  
944

```
package org.osoa.sca;  
  
public class InvalidServiceException extends ServiceRuntimeException {  
    ...  
}
```

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

## 948 **7.107.9 ConversationEndedException**

949 The following snippet shows the **ConversationEndedException**.

```
950
951     package org.osoa.sca;
952
953     public class ConversationEndedException extends ServiceRuntimeException {
954         ...
955     }
956
```

---

## 957 8 Java Annotations

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

### 959 8.1 @AllowsPassByReference

960 The following Java code defines the **@AllowsPassByReference** annotation:

```
961
962 package org.osoa.sca.annotations;
963
964 import static java.lang.annotation.ElementType.TYPE;
965 import static java.lang.annotation.ElementType.METHOD;
966 import static java.lang.annotation.RetentionPolicy.RUNTIME;
967 import java.lang.annotation.Retention;
968 import java.lang.annotation.Target;
969
970 @Target({TYPE, METHOD})
971 @Retention(RUNTIME)
972 public @interface AllowsPassByReference {
973
974 }
975
```

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

984 **@AllowsPassByReference** has no attributes

985

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

988

```
989 @AllowsPassByReference
990 public String hello(String message) {
991     ...
992 }
```

### 993 8.2 @Callback

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

995

```
996 package org.osoa.sca.annotations;
997
998 import static java.lang.annotation.ElementType.TYPE;
999 import static java.lang.annotation.ElementType.METHOD;
1000 import static java.lang.annotation.ElementType.FIELD;
1001 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1002 import java.lang.annotation.Retention;
```



```

1003     import java.lang.annotation.Target;
1004
1005     @Target(TYPE, METHOD, FIELD)
1006     @Retention(RUNTIME)
1007     public @interface Callback {
1008
1009         Class<?> value() default Void.class;
1010     }
1011
1012

```

1013 The @Callback annotation is used to annotate a service interface with a callback interface, which  
1014 takes the Java Class object of the callback interface as a parameter.

1015 The @Callback annotation has the following attribute:

- 1016 • **value** – the name of a Java class file containing the callback interface

1017

1018 The @Callback annotation may also be used to annotate a method or a field of an SCA  
1019 implementation class, in order to have a callback object injected

1020

1021 The following snippet shows a callback annotation on an interface:

1022

```

1023 @Remotable
1024 @Callback(MyServiceCallback.class)
1025 public interface MyService {
1026
1027     void someAsyncMethod(String arg);
1028 }
1029

```

1030 An example use of the @Callback annotation to declare a callback interface follows:

1031

```

1032 package somepackage;
1033 import org.osoa.sca.annotations.Callback;
1034 import org.osoa.sca.annotations.Remotable;
1035 @Remotable
1036 @Callback(MyServiceCallback.class)
1037 public interface MyService {
1038
1039     void someMethod(String arg);
1040 }
1041
1042 @Remotable
1043 public interface MyServiceCallback {
1044
1045     void receiveResult(String result);
1046 }
1047

```

1048 In this example, the implied component type is:

1049

```

1050 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >
1051
1052     <service name="MyService">
1053         <interface.java interface="somepackage.MyService"

```

```
1054         callbackInterface="somepackage.MyServiceCallback"/>
1055     </service>
1056 </componentType>
```

### 1057 8.3 @ComponentName

1058 The following Java code defines the **@ComponentName** annotation:

```
1059
1060 package org.osoa.sca.annotations;
1061
1062 import static java.lang.annotation.ElementType.METHOD;
1063 import static java.lang.annotation.ElementType.FIELD;
1064 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1065 import java.lang.annotation.Retention;
1066 import java.lang.annotation.Target;
1067
1068 @Target({METHOD, FIELD})
1069 @Retention(RUNTIME)
1070 public @interface ComponentName {
1071
1072 }
1073
```

1074 The @ComponentName annotation is used to denote a Java class field or setter method that is  
1075 used to inject the component name.

1076

1077 The following snippet shows a component name field definition sample.

1078

```
1079 @ComponentName
1080 private String componentName;
1081
```

1082 The following snippet shows a component name setter method sample.

1083

```
1084 @ComponentName
1085 public void setComponentName(String name) {
1086     //...
1087 }
```

### 1088 8.4 @Constructor

1089 The following Java code defines the **@Constructor** annotation:

1090

```
1091 package org.osoa.sca.annotations;
1092
1093 import static java.lang.annotation.ElementType.CONSTRUCTOR;
1094 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1095 import java.lang.annotation.Retention;
1096 import java.lang.annotation.Target;
1097
1098 @Target(CONSTRUCTOR)
1099 @Retention(RUNTIME)
1100 public @interface Constructor {
1101     String[] value() default "";
1102 }
```

1103  
1104  
1105  
1106  
1107  
1108  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
  
1127  
1128  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1140  
1141  
1142  
1143  
  
1144  
1145  
1146  
1147  
1148  
1149  
1150  
1151

The @Constructor annotation is used to mark a particular constructor to use when instantiating a Java component implementation.

The @Constructor annotation has the following attribute:

- **value (optional)** – identifies the property/reference names that correspond to each of the constructor arguments. The position in the array determines which of the arguments are being named.

The following snippet shows a sample for the Constructor annotation.

```
public class HelloServiceImpl implements HelloService {  
    public HelloServiceImpl(){  
        ...  
    }  
  
    @Constructor  
    public HelloServiceImpl( String someProperty ){  
        ...  
    }  
  
    public String hello(String message) {  
        ...  
    }  
}
```

## 8.5 @Context

The following Java code defines the @Context annotation:

```
package org.osoa.sca.annotations;  
  
import static java.lang.annotation.ElementType.METHOD;  
import static java.lang.annotation.ElementType.FIELD;  
import static java.lang.annotation.RetentionPolicy.RUNTIME;  
import java.lang.annotation.Retention;  
import java.lang.annotation.Target;  
  
@Target({METHOD, FIELD})  
@Retention(RUNTIME)  
public @interface Context {  
}
```

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

The @Context annotation has no attributes.

The following snippet shows a ComponentContext field definition sample.

```
1152 @Context
1153 protected ComponentContext context;
1154
```

1155 The following snippet shows a RequestContext field definition sample.

```
1156
1157 @Context
1158 protected RequestContext context;
```

## 1159 8.6 @Conversational

1160 The following Java code defines the **@Conversational** annotation:

```
1161
1162 package org.osoa.sca.annotations;
1163
1164 import static java.lang.annotation.ElementType.TYPE;
1165 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1166 import java.lang.annotation.Retention;
1167 import java.lang.annotation.Target;
1168 @Target(TYPE)
1169 @Retention(RUNTIME)
1170 public @interface Conversational {
1171 }
1172
```

1173 The @Conversational annotation is used on a Java interface to denote a conversational service  
1174 contract.

1175 The @Conversational annotation has no attributes.

1176 The following snippet shows a sample for the Conversational annotation.

```
1177 package services.hello;
1178
1179 import org.osoa.sca.annotations.Conversational;
1180
1181 @Conversational
1182 public interface HelloService {
1183     void setName(String name);
1184     String sayHello();
1185 }
```

## 1186 8.7 @ConversationAttributes

1187 The following Java code defines the **@ConversationAttributes** annotation:

```
1188
1189 package org.osoa.sca.annotations;
1190
1191 import static java.lang.annotation.ElementType.TYPE;
1192 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1193 import java.lang.annotation.Retention;
1194 import java.lang.annotation.Target;
1195
1196 @Target(TYPE)
1197 @Retention(RUNTIME)
1198 public @interface ConversationAttributes {
1199
1200     String maxIdleTime() default "";
```

```

1201     String maxAge() default "";
1202     boolean singlePrincipal() default false;
1203 }
1204

```

1205 The `@ConversationAttributes` annotation is used to define a set of attributes which apply to  
1206 conversational interfaces of services or references of a Java class. The annotation has the following  
1207 attributes:

- 1208 • ***maxIdleTime (optional)*** - The maximum time that can pass between successive  
1209 operations within a single conversation. If more time than this passes, then the container  
1210 may end the conversation.
- 1211 • ***maxAge (optional)*** - The maximum time that the entire conversation can remain active.  
1212 If more time than this passes, then the container may end the conversation.
- 1213 • ***singlePrincipal (optional)*** – If true, only the principal (the user) that started the  
1214 conversation has authority to continue the conversation. The default value is false.

1215

1216 The two attributes that take a time express the time as a string that starts with an integer, is  
1217 followed by a space and then one of the following: "seconds", "minutes", "hours", "days" or  
1218 "years".

1219

1220 Not specifying timeouts means that timeouts are defined by the SCA runtime implementation,  
1221 however it chooses to do so.

1222

1223 The following snippet shows the use of the `@ConversationAttributes` annotation to set the  
1224 maximum age for a Conversation to be 30 days.

1225

```

1226 package service.shoppingcart;
1227
1228 import org.osoa.sca.annotations.ConversationAttributes;
1229
1230 @ConversationAttributes (maxAge="30 days");
1231 public class ShoppingCartServiceImpl implements ShoppingCartService {
1232     ...
1233 }

```

## 1234 8.8 @ConversationID

1235 The following Java code defines the `@ConversationID` annotation:

1236

```

1237 package org.osoa.sca.annotations;
1238
1239 import static java.lang.annotation.ElementType.METHOD;
1240 import static java.lang.annotation.ElementType.FIELD;
1241 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1242 import java.lang.annotation.Retention;
1243 import java.lang.annotation.Target;
1244
1245 @Target({METHOD, FIELD})
1246 @Retention(RUNTIME)
1247 public @interface ConversationID {
1248
1249 }

```

1250

1251 The @ConversationID annotation is used to annotate a Java class field or setter method that is  
1252 used to inject the conversation ID. System generated conversation IDs are always strings, but  
1253 application generated conversation IDs may be other complex types.

1254 The following snippet shows a conversation ID field definition sample.

1255

```
1256 @ConversationID  
1257 private String conversationID;
```

1258

1259 The type of the field is not necessarily String.

1260

## 1261 8.9 @Destroy

1262 The following Java code defines the **@Destroy** annotation:

1263

```
1264 package org.osoa.sca.annotations;  
1265  
1266 import static java.lang.annotation.ElementType.METHOD;  
1267 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1268 import java.lang.annotation.Retention;  
1269 import java.lang.annotation.Target;  
1270  
1271 @Target(METHOD)  
1272 @Retention(RUNTIME)  
1273 public @interface Destroy {  
1274  
1275 }  
1276
```

1277 The @Destroy annotation is used to denote a single Java class method that will be called when the  
1278 scope defined for the implementation class ends. The method MAY have any access modifier and  
1279 MUST have a void return value and no arguments.

1280 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method  
1281 when the scope defined for the implementation class ends. If the implementation class has a  
1282 method with an @Destroy annotation that does not match these criteria, the SCA runtime MUST  
1283 NOT instantiate the implementation class.

1284

1285 The following snippet shows a sample for a destroy method definition.

1286

```
1287 @Destroy  
1288 void myDestroyMethod() {  
1289     ...  
1290 }
```

## 1291 8.10 @EagerInit

1292 The following Java code defines the **@EagerInit** annotation:

1293

```
1294 package org.osoa.sca.annotations;  
1295
```

```

1296     import static java.lang.annotation.ElementType.TYPE;
1297     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1298     import java.lang.annotation.Retention;
1299     import java.lang.annotation.Target;
1300
1301     @Target(TYPE)
1302     @Retention(RUNTIME)
1303     public @interface EagerInit {
1304
1305     }

```

## 1306 8.11 @EndsConversation

1307 The following Java code defines the **@EndsConversation** annotation:

1308

```

1309     package org.osoa.sca.annotations;
1310
1311     import static java.lang.annotation.ElementType.METHOD;
1312     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1313     import java.lang.annotation.Retention;
1314     import java.lang.annotation.Target;
1315
1316     @Target(METHOD)
1317     @Retention(RUNTIME)
1318     public @interface EndsConversation {
1319
1320
1321     }
1322

```

1323 The @EndsConversation annotation is used to denote a method on a Java interface that is called  
1324 to end a conversation.

1325 The @EndsConversation annotation has no attributes.

1326 The following snippet shows a sample using the @EndsConversation annotation.

```

1327     package services.shoppingbasket;
1328
1329     import org.osoa.sca.annotations.EndsConversation;
1330
1331     public interface ShoppingBasket {
1332         void addItem(String itemID, int quantity);
1333
1334         @EndsConversation
1335         void buy();
1336     }

```

## 1337 8.12 @Init

1338 The following Java code defines the **@Init** annotation:

1339

```

1340     package org.osoa.sca.annotations;
1341
1342     import static java.lang.annotation.ElementType.METHOD;
1343     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1344     import java.lang.annotation.Retention;
1345     import java.lang.annotation.Target;

```

```

1346
1347 @Target(METHOD)
1348 @Retention(RUNTIME)
1349 public @interface Init {
1350
1351
1352 }
1353

```

1354 The @Init annotation is used to denote a single Java class method that is called when the scope  
1355 defined for the implementation class starts. The method MAY have any access modifier and MUST  
1356 have a void return value and no arguments.

1357 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method  
1358 after all property and reference injection is complete. If the implementation class has a method  
1359 with an @Init annotation that does not match these criteria, the SCA runtime MUST NOT  
1360 instantiate the implementation class.

1361 The following snippet shows an example of an init method definition.

```

1362
1363 @Init
1364 public void myInitMethod() {
1365     ...
1366 }

```

## 1367 8.13 @OneWay

1368 The following Java code defines the **@OneWay** annotation:

```

1369
1370 package org.osoa.sca.annotations;
1371
1372 import static java.lang.annotation.ElementType.METHOD;
1373 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1374 import java.lang.annotation.Retention;
1375 import java.lang.annotation.Target;
1376
1377 @Target(METHOD)
1378 @Retention(RUNTIME)
1379 public @interface OneWay {
1380
1381
1382 }
1383

```

1384 The @OneWay annotation is used on a Java interface or class method to indicate that invocations  
1385 will be dispatched in a non-blocking fashion as described in the section on Asynchronous  
1386 Programming.

1387 The @OneWay annotation has no attributes.

1388 The following snippet shows the use of the @OneWay annotation on an interface.

```

1389 package services.hello;
1390
1391 import org.osoa.sca.annotations.OneWay;
1392
1393 public interface HelloService {
1394     @OneWay
1395     void hello(String name);
1396 }

```



## 1397 8.14 @Property

1398 The following Java code defines the **@Property** annotation:

1399

```
1400 package org.osoa.sca.annotations;
1401
1402 import static java.lang.annotation.ElementType.METHOD;
1403 import static java.lang.annotation.ElementType.FIELD;
1404 import static java.lang.annotation.ElementType.PARAMETER;
1405 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1406 import java.lang.annotation.Retention;
1407 import java.lang.annotation.Target;
1408
1409 @Target({METHOD, FIELD, PARAMETER})
1410 @Retention(RUNTIME)
1411 public @interface Property {
1412
1413     String name() default "";
1414     boolean required() default false;
1415 }
1416
```

1417 The @Property annotation is used to denote a Java class field or a setter method that is used to  
1418 inject an SCA property value. The type of the property injected, which can be a simple Java type  
1419 or a complex Java type, is defined by the type of the Java class field or the type of the setter  
1420 method input argument.

1421 The @Property annotation may be used on fields, on setter methods or on a constructor method  
1422 parameter.

1423 Properties may also be injected via setter methods even when the @Property annotation is not  
1424 present. However, the @Property annotation must be used in order to inject a property onto a  
1425 non-public field. In the case where there is no @Property annotation, the name of the property is  
1426 the same as the name of the field or setter.

1427 Where there is both a setter method and a field for a property, the setter method is used.

1428

1429 The @Property annotation has the following attributes:

- 1430 • **name (optional)** – the name of the property, defaults to the name of the field of the Java  
1431 class
- 1432 • **required (optional)** – specifies whether injection is required, defaults to false

1433

1434 The following snippet shows a property field definition sample.

1435

```
1436 @Property(name="currency", required=true)
1437 protected String currency;
```

1438

1439 The following snippet shows a property setter sample

1440

```
1441 @Property(name="currency", required=true)
1442 public void setCurrency( String theCurrency ) {
```

1443

```
.....
```

1444 }

1445

1446 If the property is defined as an array or as any type that extends or implements  
1447 **java.util.Collection**, then the implied component type has a property with a **many** attribute set to  
1448 true.

1449

1450 The following snippet shows the definition of a configuration property using the @Property  
1451 annotation for a collection.

1452

```
1453 ...  
1454 private List<String> helloConfigurationProperty;  
1455  
1456 @Property(required=true)  
1457 public void setHelloConfigurationProperty(List<String> property) {  
1458     helloConfigurationProperty = property;  
1459 }  
1460 ...
```

## 1461 8.15 @Reference

1462 The following Java code defines the **@Reference** annotation:

1463

```
1464 package org.osoa.sca.annotations;  
1465  
1466 import static java.lang.annotation.ElementType.METHOD;  
1467 import static java.lang.annotation.ElementType.FIELD;  
1468 import static java.lang.annotation.ElementType.PARAMETER;  
1469 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1470 import java.lang.annotation.Retention;  
1471 import java.lang.annotation.Target;  
1472 @Target({METHOD, FIELD, PARAMETER})  
1473 @Retention(RUNTIME)  
1474 public @interface Reference {  
1475  
1476     String name() default "";  
1477     boolean required() default true;  
1478 }  
1479
```

1480 The @Reference annotation is used to denote a Java class field, a setter method, or a constructor  
1481 parameter that is used to inject a service that resolves the reference. The interface of the service  
1482 injected is defined by the type of the Java class field or the type of the setter method input  
1483 argument.

1484 References may also be injected via setter methods even when the @Reference annotation is not  
1485 present. However, the @Reference annotation must be used in order to inject a reference onto a  
1486 non-public field. In the case where there is no @Reference annotation, the name of the reference  
1487 is the same as the name of the field or setter.

1488 Where there is both a setter method and a field for a reference, the setter method is used.

1489 The @Reference annotation has the following attributes:

- 1490 • **name (optional)** – the name of the reference, defaults to the name of the field of the Java  
1491 class
- 1492 • **required (optional)** – whether injection of service or services is required. Defaults to true.

1493

1494 The following snippet shows a reference field definition sample.

1495

```
1496 @Reference(name="stockQuote", required=true)
1497 protected StockQuoteService stockQuote;
```

1498

1499 The following snippet shows a reference setter sample

1500

```
1501 @Reference(name="stockQuote", required=true)
1502 public void setStockQuote( StockQuoteService theSQService );
```

1503

1504 The following fragment from a component implementation shows a sample of a service reference  
1505 using the @Reference annotation. The name of the reference is "helloService" and its type is  
1506 HelloService. The clientMethod() calls the "hello" operation of the service referenced by the  
1507 helloService reference.

1508

```
1509 package services.hello;
1510
1511 private HelloService helloService;
1512
1513 @Reference(name="helloService", required=true)
1514 public setHelloService(HelloService service) {
1515     helloService = service;
1516 }
1517
1518 public void clientMethod() {
1519     String result = helloService.hello("Hello World!");
1520     ...
1521 }
1522
```

1523 The presence of a @Reference annotation is reflected in the componentType information that the  
1524 runtime generates through reflection on the implementation class. The following snippet shows  
1525 the component type for the above component implementation fragment.

1526

```
1527 <?xml version="1.0" encoding="ASCII"?>
1528 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1529
1530     <!-- Any services offered by the component would be listed here -->
1531     <reference name="helloService" multiplicity="1..1">
1532         <interface.java interface="services.hello.HelloService"/>
1533     </reference>
1534
1535 </componentType>
```

1536

1537 If the reference is not an array or collection, then the implied component type has a reference  
1538 with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference **required**  
1539 attribute – 1..1 applies if required=true.

1540

1541 If the reference is defined as an array or as any type that extends or implements **java.util.Collection**,  
1542 then the implied component type has a reference with a **multiplicity** of either **1..n** or **0..n**, depending

1543 on whether the **required** attribute of the **@Reference** annotation is set to true or false – 1..n applies if  
1544 required=true.

1545  
1546 The following fragment from a component implementation shows a sample of a service reference  
1547 definition using the **@Reference** annotation on a `java.util.List`. The name of the reference is  
1548 "helloServices" and its type is `HelloService`. The `clientMethod()` calls the "hello" operation of all the  
1549 services referenced by the `helloServices` reference. In this case, at least one `HelloService` should  
1550 be present, so **required** is true.

```
1551     @Reference(name="helloServices", required=true)
1552     protected List<HelloService> helloServices;
1553
1554     public void clientMethod() {
1555         ...
1556         for (int index = 0; index < helloServices.size(); index++) {
1557             HelloService helloService =
1558                 (HelloService)helloServices.get(index);
1559             String result = helloService.hello("Hello World!");
1560         }
1561     }
1562     ...
1563 }
1564
1565
```

1566 The following snippet shows the XML representation of the component type reflected from for the  
1567 former component implementation fragment. There is no need to author this component type in  
1568 this case since it can be reflected from the Java class.

```
1569
1570 <?xml version="1.0" encoding="ASCII"?>
1571 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1572
1573     <!-- Any services offered by the component would be listed here -->
1574     <reference name="helloServices" multiplicity="1..n">
1575         <interface.java interface="services.hello>HelloService"/>
1576     </reference>
1577
1578 </componentType>
1579
```

1580 At runtime, the representation of an unwired reference depends on the reference's multiplicity. An  
1581 unwired reference with a multiplicity of 0..1 must be null. An unwired reference with a multiplicity  
1582 of 0..N must be an empty array or collection.

## 1583 8.15.1 Reinjection

1584 References MAY be reinjected after the initial creation of a component if the reference target  
1585 changes due to a change in wiring that has occurred since the component was initialized. In order  
1586 for reinjection to occur, the following MUST be true:

- 1587 1. The component MUST NOT be STATELESS or REQUEST scoped.
- 1588 2. The reference MUST use either field-based injection or setter injection. References that are  
1589 injected through constructor injection MUST NOT be changed. Setter injection allows for  
1590 code in the setter method to perform processing in reaction to a change.
- 1591 3. If the reference has a conversational interface, then reinjection MUST NOT occur while the  
1592 conversation is active.

1593 If a reference target changes and the reference is not reinjected, the reference MUST continue to  
1594 work as if the reference target was not changed.

1595 If an operation is called on a reference where the target of that reference has been undeployed,  
 1596 the SCA runtime SHOULD throw InvalidServiceException. If an operation is called on a reference  
 1597 where the target of the reference has become unavailable for some reason, the SCA runtime  
 1598 SHOULD throw ServiceUnavailableException. If the target of the reference is changed, the  
 1599 reference MAY continue to work, depending on the runtime and the type of change that was made.  
 1600 If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.

1601 A ServiceReference that has been obtained from a reference by ComponentContext.cast()  
 1602 corresponds to the reference that is passed as a parameter to cast(). If the reference is  
 1603 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue  
 1604 to work as if the reference target was not changed. If the target of a ServiceReference has been  
 1605 undeployed, the SCA runtime SHOULD throw InvalidServiceException when an operation is  
 1606 invoked on the ServiceReference. If the target of a ServiceReference has become unavailable, the  
 1607 SCA runtime SHOULD throw ServiceUnavailableException when an operation is invoked on the  
 1608 ServiceReference. If the target of a ServiceReference is changed, the reference MAY continue to  
 1609 work, depending on the runtime and the type of change that was made. If it doesn't work, the  
 1610 exception thrown will depend on the runtime and the cause of the failure.

1611 A reference or ServiceReference accessed through the component context by calling getService()  
 1612 or getServiceReference() MUST correspond to the current configuration of the domain. This  
 1613 applies whether or not reinjection has taken place. If the target has been undeployed or has  
 1614 become unavailable, the result SHOULD be a reference to the undeployed or unavailable service,  
 1615 and attempts to call business methods SHOULD throw an exception as described above. If the  
 1616 target has changed, the result SHOULD be a reference to the changed service.

1617 The rules for reference reinjection also apply to references with a multiplicity of 0..N or 1..N. This  
 1618 means that in the cases listed above where reference reinjection is not allowed, the array or  
 1619 Collection for the reference MUST NOT change its contents. In cases where the contents of a  
 1620 reference collection MAY change, then for references that use setter injection, the setter method  
 1621 MUST be called for any change to the contents. The reinjected array or Collection MUST NOT be  
 1622 the same array or Collection object previously injected to the component.

1623

<u>Change event</u>	<u>Effect on</u>		
	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	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	Result SHOULD be a reference to the changed service.

	failure.	failure.	
<p>* Other conditions:</p> <ol style="list-style-type: none"> <li>1. The component MUST NOT be STATELESS or REQUEST 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>			

1624

## 1625 8.16 @Remotable

1626 The following Java code defines the **@Remotable** annotation:

1627

```

1628 package org.osoa.sca.annotations;
1629
1630 import static java.lang.annotation.ElementType.TYPE;
1631 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1632 import java.lang.annotation.Retention;
1633 import java.lang.annotation.Target;
1634
1635
1636 @Target(TYPE)
1637 @Retention(RUNTIME)
1638 public @interface Remotable {
1639
1640 }
1641

```

1642 The @Remotable annotation is used to specify a Java service interface as remotable. A remotable  
1643 service can be published externally as a service and must be translatable into a WSDL portType.

1644 The @Remotable annotation has no attributes.

1645

1646 The following snippet shows the Java interface for a remotable service with its @Remotable  
1647 annotation.

```

1648 package services.hello;
1649
1650 import org.osoa.sca.annotations.*;
1651
1652 @Remotable
1653 public interface HelloService {
1654
1655     String hello(String message);
1656 }
1657

```

1658 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**  
1659 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

1660

1661 Complex data types exchanged via remotable service interfaces must be compatible with the  
1662 marshalling technology used by the service binding. For example, if the service is going to be  
1663 exposed using the standard web service binding, then the parameters must be Service Data  
1664 Objects (SDOs) 2.0 or 2.1 [2] or JAXB 2.0 [3] types.

1665 Independent of whether the remotable service is called from outside of the composite that  
1666 contains it or from another component in the same composite, the data exchange semantics are  
1667 **by-value**.

1668 Implementations of remotable services may modify input data during or after an invocation and  
1669 may modify return data after the invocation. If a remotable service is called locally or remotely,  
1670 the SCA container is responsible for making sure that no modification of input data or post-  
1671 invocation modifications to return data are seen by the caller.

1672

1673 The following snippet shows a remotable Java service interface.

1674

```
1675 package services.hello;
1676
1677 import org.osoa.sca.annotations.*;
1678
1679 @Remotable
1680 public interface HelloService {
1681     String hello(String message);
1682 }
1683
1684 package services.hello;
1685
1686 import org.osoa.sca.annotations.*;
1687
1688 @Service(HelloService.class)
1689 public class HelloServiceImpl implements HelloService {
1690     public String hello(String message) {
1691         ...
1692     }
1693 }
1694
1695 }
```

## 1696 8.17 @Scope

1697 The following Java code defines the **@Scope** annotation:

1698

```
1699 package org.osoa.sca.annotations;
1700
1701 import static java.lang.annotation.ElementType.TYPE;
1702 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1703 import java.lang.annotation.Retention;
1704 import java.lang.annotation.Target;
1705
1706 @Target(TYPE)
1707 @Retention(RUNTIME)
1708 public @interface Scope {
1709     String value() default "STATELESS";
1710 }
1711 }
```

1712 The @Scope annotation may only be used on a service's implementation class. It is an error to use  
1713 this annotation on an interface.

1714 The @Scope annotation has the following attribute:

- 1715 • **value** – the name of the scope.  
1716 The default value is 'STATELESS'. For 'STATELESS' implementations, a different

1717 implementation instance may be used to service each request. Implementation instances  
1718 may be newly created or be drawn from a pool of instances.  
1719 SCA defines the following scope names, but others can be defined by particular Java-  
1720 based implementation types:  
1721 STATELESS  
1722 REQUEST  
1723 COMPOSITE  
1724 CONVERSATION

1725 The following snippet shows a sample for a CONVERSATION scoped service implementation:

```
1726 package services.hello;
1727
1728 import org.osoa.sca.annotations.*;
1729
1730 @Service(HelloService.class)
1731 @Scope("CONVERSATION")
1732 public class HelloServiceImpl implements HelloService {
1733
1734     public String hello(String message) {
1735         ...
1736     }
1737 }
1738
```

## 1739 8.18 @Service

1740 The following Java code defines the **@Service** annotation:

```
1741
1742 package org.osoa.sca.annotations;
1743
1744 import static java.lang.annotation.ElementType.TYPE;
1745 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1746 import java.lang.annotation.Retention;
1747 import java.lang.annotation.Target;
1748
1749 @Target(TYPE)
1750 @Retention(RUNTIME)
1751 public @interface Service {
1752
1753     Class<?>[] interfaces() default {};
1754     Class<?> value() default Void.class;
1755 }
1756
```

1757 The @Service annotation is used on a component implementation class to specify the SCA services  
1758 offered by the implementation. The class need not be declared as implementing all of the  
1759 interfaces implied by the services, but all methods of the service interfaces must be present. A  
1760 class used as the implementation of a service is not required to have an @Service annotation. If a  
1761 class has no @Service annotation, then the rules determining which services are offered and what  
1762 interfaces those services have are determined by the specific implementation type.

1763 The @Service annotation has the following attributes:

- 1764 • **interfaces** – The value is an array of interface or class objects that should be exposed as  
1765 services by this component.
- 1766 • **value** – A shortcut for the case when the class provides only a single service interface.

1767 Only one of these attributes should be specified.



1768

1769 A @Service annotation with no attributes is meaningless, it is the same as not having the  
1770 annotation there at all.

1771 The **service names** of the defined services default to the names of the interfaces or class, without  
1772 the package name.

1773 If a Java implementation needs to realize two services with the same interface, then this is  
1774 achieved through subclassing of the interface. The subinterface must not add any methods. Both  
1775 interfaces are listed in the @Service annotation of the Java implementation class.

1776 The following snippet shows an implementation of the HelloService marked with the @Service  
1777 annotation.

```
1778 package services.hello;  
1779  
1780 import org.oesa.sca.annotations.Service;  
1781  
1782 @Service(HelloService.class)  
1783 public class HelloServiceImpl implements HelloService {  
1784  
1785     public void hello(String name) {  
1786         System.out.println("Hello " + name);  
1787     }  
1788 }  
1789
```

1790

## 9 WSDL to Java and Java to WSDL

1791 The SCA Client and Implementation Model for Java applies the WSDL to Java and Java to WSDL  
1792 mapping rules as defined by the JAX-WS specification [7] for generating remotable Java interfaces  
1793 from WSDL portTypes and vice versa.

1794 For the purposes of the Java-to-WSDL mapping algorithm, the interface is treated as if it had a  
1795 @WebService annotation on the class, even if it doesn't, and the org.osoa.annotations.OneWay  
1796 annotation should be treated as a synonym for javax.jws.OneWay. For the WSDL-to-Java  
1797 mapping, the generated @WebService annotation implies that the interface is @Remotable.

1798 For the mapping from Java types to XML schema types SCA supports both the SDO 2.1 [2]  
1799 mapping and the JAXB 2.0 [3] mapping. Having a choice of binding technologies is allowed, as  
1800 noted in the first paragraph of section 5 of the JSR 181 (version 2) specification, which is  
1801 referenced by the JAX-WS specification.

1802 The JAX-WS mappings are applied with the following restrictions:

- 1803 • No support for holders

1804

1805 **Note:** This specification needs more examples and discussion of how JAX-WS's client asynchronous  
1806 model is used.

### 1807 9.1 JAX-WS Client Asynchronous API for a Synchronous Service

1808 The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client  
1809 application with a means of invoking that service asynchronously, so that the client can invoke a service  
1810 operation and proceed to do other work without waiting for the service operation to complete its  
1811 processing. The client application can retrieve the results of the service either through a polling  
1812 mechanism or via a callback method which is invoked when the operation completes.

1813 For SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional  
1814 client-side asynchronous polling and callback methods defined by JAX-WS. For SCA service interfaces  
1815 defined using interface.java, the Java interface MUST NOT contain these methods. If these methods are  
1816 present, SCA Runtimes MUST NOT include them in the SCA reference interface as defined by the  
1817 Assembly specification. These methods are recognized as follows.

1818 For each method M in the interface, if another method P in the interface has

- 1819 a. a method name that is M's method name with the characters "Async" appended, and
- 1820 b. the same parameter signature as M, and
- 1821 c. a return type of Response<R> where R is the return type of M

1822 then P is a JAX-WS polling method that isn't part of the SCA interface contract.

1823 For each method M in the interface, if another method C in the interface has

- 1824 a. a method name that is M's method name with the characters "Async" appended, and
- 1825 b. a parameter signature that is M's parameter signature with an additional final parameter of type  
1826 AsyncHandler<R> where R is the return type of M, and
- 1827 c. a return type of Future<?>

1828 then C is a JAX-WS callback method that isn't part of the SCA interface contract.

1829 As an example, an interface may be defined in WSDL as follows:

```
1830 <!-- WSDL extract -->  
1831 <message name="getPrice">  
1832   <part name="ticker" type="xsd:string"/>  
1833 </message>  
1834
```

```
1835 <message name="getPriceResponse">
1836   <part name="price" type="xsd:float"/>
1837 </message>
1838
1839 <portType name="StockQuote">
1840   <operation name="getPrice">
1841     <input message="tns:getPrice"/>
1842     <output message="tns:getPriceResponse"/>
1843   </operation>
1844 </portType>
```

1845

1846 The JAX-WS asynchronous mapping will produce the following Java interface:

```
1847 // asynchronous mapping
1848 @WebService
1849 public interface StockQuote {
1850   float getPrice(String ticker);
1851   Response<Float> getPriceAsync(String ticker);
1852   Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
1853 }
```

1854

1855 For SCA interface definition purposes, this is treated as equivalent to the following:

```
1856 // synchronous mapping
1857 @WebService
1858 public interface StockQuote {
1859   float getPrice(String ticker);
1860 }
```

1861

1862 SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above  
1863 example, if the client implementation uses the asynchronous form of the interface, the two  
1864 additional getPriceAsync() methods can be used for polling and callbacks as defined by the JAX-  
1865 WS specification.

---

## 1866 10 Policy Annotations for Java

1867 SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which  
1868 influence how implementations, services and references behave at runtime. The policy facilities  
1869 are described in [the SCA Policy Framework specification \[5\]](#). In particular, the facilities include  
1870 Intents and Policy Sets, where intents express abstract, high-level policy requirements and policy  
1871 sets express low-level detailed concrete policies.

1872 Policy metadata can be added to SCA assemblies through the means of declarative statements  
1873 placed into Composite documents and into Component Type documents. These annotations are  
1874 completely independent of implementation code, allowing policy to be applied during the assembly  
1875 and deployment phases of application development.

1876 However, it can be useful and more natural to attach policy metadata directly to the code of  
1877 implementations. This is particularly important where the policies concerned are relied on by the  
1878 code itself. An example of this from the Security domain is where the implementation code  
1879 expects to run under a specific security Role and where any service operations invoked on the  
1880 implementation must be authorized to ensure that the client has the correct rights to use the  
1881 operations concerned. By annotating the code with appropriate policy metadata, the developer  
1882 can rest assured that this metadata is not lost or forgotten during the assembly and deployment  
1883 phases.

1884 The SCA Java Common Annotations specification provides a series of annotations which provide  
1885 the capability for the developer to attach policy information to Java implementation code. The  
1886 annotations concerned first provide general facilities for attaching SCA Intents and Policy Sets to  
1887 Java code. Secondly, there are further specific annotations that deal with particular policy intents  
1888 for certain policy domains such as Security.

1889 The SCA Java Common Annotations specification supports using [the Common Annotation for Java  
1890 Platform specification \(JSR-250\) \[6\]](#). An implication of adopting the common annotation for Java  
1891 platform specification is that the SCA Java specification support consistent annotation and Java  
1892 class inheritance relationships.

1893

### 1894 10.1 General Intent Annotations

1895 SCA provides the annotation **@Requires** for the attachment of any intent to a Java class, to a  
1896 Java interface or to elements within classes and interfaces such as methods and fields.

1897 The @Requires annotation can attach one or multiple intents in a single statement.

1898 Each intent is expressed as a string. Intents are XML QNames, which consist of a Namespace URI  
1899 followed by the name of the Intent. The precise form used follows the string representation used  
1900 by the `javax.xml.namespace.QName` class, which is as follows:

1901 `"{" + Namespace URI + "}" + intentname`

1902 Intents may be qualified, in which case the string consists of the base intent name, followed by a  
1903 ".", followed by the name of the qualifier. There may also be multiple levels of qualification.

1904 This representation is quite verbose, so we expect that reusable String constants will be defined  
1905 for the namespace part of this string, as well as for each intent that is used by Java code. SCA  
1906 defines constants for intents such as the following:

```
1907 public static final String SCA_PREFIX="{http://docs.oasis-  
1908 open.org/ns/opencsa/sca/200712}";
```

```
1909 public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
```

```
1910 public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
```

1911 Notice that, by convention, qualified intents include the qualifier as part of the name of the  
1912 constant, separated by an underscore. These intent constants are defined in the file that defines  
1913 an annotation for the intent (annotations for intents, and the formal definition of these constants,  
1914 are covered in a following section).

1915 Multiple intents (qualified or not) are expressed as separate strings within an array declaration.

1916 An example of the @Requires annotation with 2 qualified intents (from the Security domain)  
1917 follows:

```
1918  
1919     @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

1920  
1921 This attaches the intents "confidentiality.message" and "integrity.message".

1922 The following is an example of a reference requiring support for confidentiality:

```
1923     package org.osoa.sca.annotation;  
1924  
1925     import static org.osoa.sca.annotation.Confidentiality.*;  
1926  
1927     public class Foo {  
1928         @Requires(CONFIDENTIALITY)  
1929         @Reference  
1930         public void setBar(Bar bar) {  
1931             ...  
1932         }  
1933     }
```

1934 Users may also choose to only use constants for the namespace part of the QName, so that they  
1935 may add new intents without having to define new constants. In that case, this definition would  
1936 instead look like this:

```
1937     package org.osoa.sca.annotation;  
1938  
1939     import static org.osoa.sca.Constants.*;  
1940  
1941     public class Foo {  
1942         @Requires(SCA_PREFIX+"confidentiality")  
1943         @Reference  
1944         public void setBar(Bar bar) {  
1945             ...  
1946         }  
1947     }
```

1948  
1949 The formal syntax for the @Requires annotation follows:

```
1950     @Requires( "qualifiedIntent" | { "qualifiedIntent" [, "qualifiedIntent"] }
```

1951 where

1952 qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2

1953

1954 The following shows the formal definition of the @Requires annotation:

1955

```
1956 package org.osoa.sca.annotation;
1957 import static java.lang.annotation.ElementType.TYPE;
1958 import static java.lang.annotation.ElementType.METHOD;
1959 import static java.lang.annotation.ElementType.FIELD;
1960 import static java.lang.annotation.ElementType.PARAMETER;
1961 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1962 import java.lang.annotation.Retention;
1963 import java.lang.annotation.Target;
1964 import java.lang.annotation.Inherited;
```

1965

```
1966 @Inherited
1967 @Retention(RUNTIME)
1968 @Target({TYPE, METHOD, FIELD, PARAMETER})
```

1969

```
1970 public @interface Requires {
1971     String[] value() default "";
1972 }
```

1973 The SCA\_NS constant is defined in the Constants interface:

```
1974 package org.osoa.sca;
1975
1976 public interface Constants {
1977     String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";
1978     String SCA_PREFIX = "{"+SCA_NS+"}";
1979 }
```

1980

## 1981 10.2 Specific Intent Annotations

1982 In addition to the general intent annotation supplied by the @Requires annotation described  
1983 above, it is also possible to have Java annotations that correspond to specific policy intents. SCA  
1984 provides a number of these specific intent annotations and it is also possible to create new specific  
1985 intent annotations for any intent.

1986 The general form of these specific intent annotations is an annotation with a name derived from  
1987 the name of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an  
1988 attribute to the annotation in the form of a string or an array of strings.

1989 For example, the SCA confidentiality intent described in [the section on General Intent Annotations](#)  
1990 using the @Requires(CONFIDENTIALITY) intent can also be specified with the specific  
1991 @Confidentiality intent annotation. The specific intent annotation for the "integrity" security intent  
1992 is:

1993           @Integrity

1994           An example of a qualified specific intent for the "authentication" intent is:

1995           @Authentication( {"message", "transport"} )

1996           This annotation attaches the pair of qualified intents: "authentication.message" and  
1997           "authentication.transport" (the sca: namespace is assumed in this both of these cases –  
1998           "http://docs.oasis-open.org/ns/opencsa/sca/200712").

1999           The general form of specific intent annotations is:

2000           @<Intent>[(qualifiers)]

2001           where Intent is an NCName that denotes a particular type of intent.

2002           Intent ::= NCName

2003           qualifiers ::= "qualifier" | { "qualifier" [, "qualifier"] }

2004           qualifier ::= NCName | NCName/qualifier

2005

## 2006   10.2.1 How to Create Specific Intent Annotations

2007           SCA identifies annotations that correspond to intents by providing an @Intent annotation which  
2008           must be used in the definition of an intent annotation.

2009           The @Intent annotation takes a single parameter, which (like the @Requires annotation) is the  
2010           String form of the QName of the intent. As part of the intent definition, it is good practice  
2011           (although not required) to also create String constants for the Namespace, the Intent and for  
2012           Qualified versions of the Intent (if defined). These String constants are then available for use with  
2013           the @Requires annotation and it should also be possible to use one or more of them as  
2014           parameters to the @Intent annotation.

2015           Alternatively, the QName of the intent may be specified using separate parameters for the  
2016           targetNamespace and the localPart for example:

2017           @Intent(targetNamespace=SCA\_NS, localPart="confidentiality").

2018           The definition of the @Intent annotation is the following:

2019

```

2020           package org.osoa.sca.annotation;
2021           import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
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(RUNTIME)
2028           @Target(ANNOTATION_TYPE)
2029           public @interface Intent {
2030               String value() default "";
2031               String targetNamespace() default "";
2032               String localPart() default "";
2033           }
```

2034 When an intent can be qualified, it is good practice for the first attribute of the annotation to be a  
2035 string (or an array of strings) which holds one or more qualifiers.

2036 In this case, the attribute's definition should be marked with the `@Qualifier` annotation. The  
2037 `@Qualifier` tells SCA that the value of the attribute should be treated as a qualifier for the intent  
2038 represented by the whole annotation. If more than one qualifier value is specified in an  
2039 annotation, it means that multiple qualified forms are required. For example:

```
2040 @Confidentiality( {"message", "transport" } )
```

2041 implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport"  
2042 are set for the element to which the confidentiality intent is attached.

2043 The following is the definition of the `@Qualifier` annotation.

2044

```
2045 package org.osoa.sca.annotation;  
2046 import static java.lang.annotation.ElementType.METHOD;  
2047 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2048 import java.lang.annotation.Retention;  
2049 import java.lang.annotation.Target;  
2050 import java.lang.annotation.Inherited;
```

2051

```
2052 @Retention(RetentionPolicy.RUNTIME)
```

```
2053 @Target( ElementType.METHOD)
```

```
2054 public @interface Qualifier {  
2055 }
```

2056

2057 Examples of the use of the `@Intent` and `@Qualifier` annotations in the definition of specific intent  
2058 annotations are shown in [the section dealing with Security Interaction Policy](#).

2059

## 2060 10.3 Application of Intent Annotations

2061 The SCA Intent annotations can be applied to the following Java elements:

- 2062 • Java class
- 2063 • Java interface
- 2064 • Method
- 2065 • Field

2066 Where multiple intent annotations (general or specific) are applied to the same Java element, they  
2067 are additive in effect. An example of multiple policy annotations being used together follows:

```
2068 @Authentication  
2069 @Requires( {CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE} )
```

2070 In this case, the effective intents are "authentication", "confidentiality.message" and  
2071 "integrity.message".

2072 If an annotation is specified at both the class/interface level and the method or field level, then  
2073 the method or field level annotation completely overrides the class level annotation of the same  
2074 type.



2075 The intent annotation can be applied either to classes or to class methods when adding annotated  
2076 policy on SCA services. Applying an intent to the setter method in a reference injection approach  
2077 allows intents to be defined at references.

### 2078 10.3.1 Inheritance And Annotation

2079 The inheritance rules for annotations are consistent with the common annotation specification, JSR  
2080 250.

2081 The following example shows the inheritance relations of intents on classes, operations, and super  
2082 classes.

```
2083
2084     package services.hello;
2085     import org.osoa.sca.annotations.Remotable;
2086     import org.osoa.sca.annotations.Integrity;
2087     import org.osoa.sca.annotations.Authentication;
2088
2089     @Integrity("transport")
2090     @Authentication
2091     public class HelloService {
2092         @Integrity
2093         @Authentication("message")
2094         public String hello(String message) {...}
2095
2096         @Integrity
2097         @Authentication("transport")
2098         public String helloThere() {...}
2099     }
2100
2101     package services.hello;
2102     import org.osoa.sca.annotations.Remotable;
2103     import org.osoa.sca.annotations.Confidentiality;
2104     import org.osoa.sca.annotations.Authentication;
2105
2106     @Confidentiality("message")
2107     public class HelloChildService extends HelloService {
2108         @Confidentiality("transport")
2109         public String hello(String message) {...}
2110         @Authentication
2111         String helloWorld() {...}
2112     }
```

2113 Example 2a. Usage example of annotated policy and inheritance.

2114

2115 The effective intent annotation on the helloWorld method is Integrity("transport"),  
2116 @Authentication, and @Confidentiality("message").

2117 The effective intent annotation on the hello method of the HelloChildService is  
 2118 @Integrity("transport"), @Authentication, and @Confidentiality("transport"),  
 2119 The effective intent annotation on the helloThere method of the HelloChildService is @Integrity  
 2120 and @Authentication("transport"), the same as in HelloService class.  
 2121 The effective intent annotation on the hello method of the HelloService is @Integrity and  
 2122 @Authentication("message")

2123

2124 The listing below contains the equivalent declarative security interaction policy of the HelloService  
 2125 and HelloChildService implementation corresponding to the Java interfaces and classes shown in  
 2126 Example 2a.

2127

```

2128 <?xml version="1.0" encoding="ASCII"?>
2129
2130 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2131           name="HelloServiceComposite" >
2132   <service name="HelloService" requires="integrity/transport
2133           authentication">
2134     ...
2135   </service>
2136   <service name="HelloChildService" requires="integrity/transport
2137           authentication confidentiality/message">
2138     ...
2139   </service>
2140   ...
2141
2142   <component name="HelloServiceComponent">*
2143     <implementation.java class="services.hello.HelloService" />
2144     <operation name="hello" requires="integrity
2145           authentication/message" />
2146     <operation name="helloThere"
2147 requires="integrity
2148           authentication/transport" />
2149   </component>
2150   <component name="HelloChildServiceComponent">*
2151     <implementation.java
2152 class="services.hello.HelloChildService" />
2153     <operation name="hello"
2154 requires="confidentiality/transport" />
2155     <operation name="helloThere" requires=" integrity/transport
2156           authentication" />
2157     <operation name="helloWorld" requires="authentication" />
2158   </component>
2159   ...
2160   ...
2161 </composite>
  
```

2162 Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.

2163

## 2166 10.4 Relationship of Declarative And Annotated Intents

2167 Annotated intents on a Java class cannot be overridden by declarative intents either in a  
 2168 composite document which uses the class as an implementation or by statements in a component

2169 Type document associated with the class. This rule follows the general rule for intents that they  
2170 represent fundamental requirements of an implementation.

2171 An unqualified version of an intent expressed through an annotation in the Java class may be  
2172 qualified by a declarative intent in a using composite document.

2173

## 2174 10.5 Policy Set Annotations

2175 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies (for  
2176 example, a concrete policy is the specific encryption algorithm to use when encrypting messages  
2177 when using a specific communication protocol to link a reference to a service).

2178 Policy Sets can be applied directly to Java implementations using the **@PolicySets** annotation.  
2179 The PolicySets annotation either takes the QName of a single policy set as a string or the name of  
2180 two or more policy sets as an array of strings:  
2181

```
2182     @PolicySets( "<policy set QName>" |  
2183                 { "<policy set QName>" [, "<policy set QName>" ] })
```

2184

2185 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

2186 An example of the @PolicySets annotation:

2187

```
2188     @Reference(name="helloService", required=true)  
2189     @PolicySets({ MY_NS + "WS_Encryption_Policy",  
2190                 MY_NS + "WS_Authentication_Policy" })  
2191     public setHelloService>HelloService service) {  
2192         . . .  
2193     }
```

2194 In this case, the Policy Sets WS\_Encryption\_Policy and WS\_Authentication\_Policy are applied, both  
2195 using the namespace defined for the constant MY\_NS.

2196 PolicySets must satisfy intents expressed for the implementation when both are present, according  
2197 to the rules defined in [the Policy Framework specification \[5\]](#).

2198 The SCA Policy Set annotation can be applied to the following Java elements:

- 2199 • Java class
- 2200 • Java interface
- 2201 • Method
- 2202 • Field

2203

## 2204 10.6 Security Policy Annotations

2205 This section introduces annotations for SCA's security intents, as defined in [the SCA Policy](#)  
2206 [Framework specification \[5\]](#).

2207

### 2208 10.6.1 Security Interaction Policy

2209 The following interaction policy Intents and qualifiers are defined for Security Policy, which apply  
2210 to the operation of services and references of an implementation:

- 2211 • @Integrity
- 2212 • @Confidentiality
- 2213 • @Authentication

2214 All three of these intents have the same pair of Qualifiers:

- 2215 • message
- 2216 • transport

2217 The following snippets shows the @Integrity, @Confidentiality and @Authentication annotations:

```

2218 package org.oesa.sca.annotation;
2219
2220 import java.lang.annotation.*;
2221 import static org.oesa.sca.Constants.SCA_NS;
2222
2223 @Inherited
2224 @Retention(RetentionPolicy.RUNTIME)
2225 @Target({ElementType.TYPE, ElementType.METHOD,
2226          ElementType.FIELD, ElementType.PARAMETER})
2227 @Intent(Integrity.INTEGRITY)
2228 public @interface Integrity {
2229     String INTEGRITY = SCA_NS+"integrity";
2230     String INTEGRITY_MESSAGE = INTEGRITY+".message";
2231     String INTEGRITY_TRANSPORT = INTEGRITY+".transport";
2232     @Qualifier
2233     String[] value() default "";
2234 }
2235
2236
2237 package org.oesa.sca.annotation;
2238
2239 import java.lang.annotation.*;
2240 import static org.oesa.sca.Constants.SCA_NS;
2241
2242 @Inherited
2243 @Retention(RetentionPolicy.RUNTIME)
2244 @Target({ElementType.TYPE, ElementType.METHOD,
2245          ElementType.FIELD, ElementType.PARAMETER})
2246 @Intent(Confidentiality.CONFIDENTIALITY)
2247 public @interface Confidentiality {
2248     String CONFIDENTIALITY = SCA_NS+"confidentiality";
2249     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY+".message";

```

```

2250         String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY+".transport";
2251         @Qualifier
2252         String[] value() default "";
2253     }
2254
2255
2256     package org.osoa.sca.annotation;
2257
2258     import java.lang.annotation.*;
2259     import static org.osoa.sca.Constants.SCA_NS;
2260
2261     @Inherited
2262     @Retention(RetentionPolicy.RUNTIME)
2263     @Target({ElementType.TYPE, ElementType.METHOD,
2264             ElementType.FIELD, ElementType.PARAMETER})
2265     @Intent(Authentication.AUTHENTICATION)
2266     public @interface Authentication {
2267         String AUTHENTICATION = SCA_NS+"authentication";
2268         String AUTHENTICATION_MESSAGE = AUTHENTICATION+".message";
2269         String AUTHENTICATION_TRANSPORT = AUTHENTICATION+".transport";
2270         @Qualifier
2271         String[] value() default "";
2272     }
2273
2274

```

2275 The following example shows an example of applying an intent to the setter method used to inject  
2276 a reference. Accessing the hello operation of the referenced HelloService requires both  
2277 "integrity.message" and "authentication.message" intents to be honored.

```

2278
2279     //Interface for HelloService
2280     public interface service.hello.HelloService {
2281         String hello(String helloMsg);
2282     }
2283
2284     // Interface for ClientService
2285     public interface service.client.ClientService {
2286         public void clientMethod();
2287     }
2288
2289     // Implementation class for ClientService
2290     package services.client;

```

```

2291
2292     import services.hello.HelloService;
2293
2294     import org.osoa.sca.annotations.*;
2295
2296     @Service(ClientService.class)
2297     public class ClientServiceImpl implements ClientService {
2298
2299
2300         private HelloService helloService;
2301
2302         @Reference(name="helloService", required=true)
2303         @Integrity("message")
2304         @Authentication("message")
2305         public void setHelloService(HelloService service) {
2306             helloService = service;
2307         }
2308
2309         public void clientMethod() {
2310             String result = helloService.hello("Hello World!");
2311             ...
2312         }
2313     }
2314

```

2315 Example 1. Usage of annotated intents on a reference.

2316

## 2317 10.6.2 Security Implementation Policy

2318 SCA defines a number of security policy annotations that apply as policies to implementations  
2319 themselves. These annotations mostly have to do with authorization and security identity. The  
2320 following authorization and security identity annotations (as defined in JSR 250) are supported:

- 2321 • RunAs

2322

2323 Takes as a parameter a string which is the name of a Security role.  
2324 eg. @RunAs("Manager")

- 2325 • Code marked with this annotation will execute with the Security permissions of the  
2326 identified role.

- 2327 • RolesAllowed

2328

2329 Takes as a parameter a single string or an array of strings which represent one or more  
2330 role names. When present, the implementation can only be accessed by principals whose  
2331 role corresponds to one of the role names listed in the @roles attribute. How role names  
2332 are mapped to security principals is implementation dependent (SCA does not define this).  
2333 eg. @RolesAllowed( {"Manager", "Employee"} )

- 2334 • PermitAll

2335

2336 No parameters. When present, grants access to all roles.

- 2337 • DenyAll
- 2338
- 2339 No parameters. When present, denies access to all roles.
- 2340 • DeclareRoles
- 2341 Takes as a parameter a string or an array of strings which identify one or more role names
- 2342 that form the set of roles used by the implementation.
- 2343 eg. @DeclareRoles({"Manager", "Employee", "Customer"} )
- 2344 (all these are declared in the Java package javax.annotation.security)
- 2345 For a full explanation of these intents, see [the Policy Framework specification \[5\]](#).

### 2346 10.6.2.1 Annotated Implementation Policy Example

2347 The following is an example showing annotated security implementation policy:

```
2348
2349 package services.account;
2350 @Remotable
2351 public interface AccountService {
2352     AccountReport getAccountReport(String customerID);
2353 }
```

2354

2355 The following is a full listing of the AccountServiceImpl class, showing the Service it implements,

2356 plus the service references it makes and the settable properties that it has, along with a set of

2357 implementation policy annotations:

```
2358
2359 package services.account;
2360 import java.util.List;
2361 import commonj.sdo.DataFactory;
2362 import org.osoa.sca.annotations.Property;
2363 import org.osoa.sca.annotations.Reference;
2364 import org.osoa.sca.annotations.RolesAllowed;
2365 import org.osoa.sca.annotations.RunAs;
2366 import org.osoa.sca.annotations.PermitAll;
2367 import services.accountdata.AccountDataService;
2368 import services.accountdata.CheckingAccount;
2369 import services.accountdata.SavingsAccount;
2370 import services.accountdata.StockAccount;
2371 import services.stockquote.StockQuoteService;
2372 @RolesAllowed("customers")
2373 @RunAs("accountants" )
2374 public class AccountServiceImpl implements AccountService {
2375
2376     @Property
2377     protected String currency = "USD";
2378
2379     @Reference
2380     protected AccountDataService accountDataService;
```

```

2381     @Reference
2382     protected StockQuoteService stockQuoteService;
2383
2384     @RolesAllowed({"customers", "accountants"})
2385     public AccountReport getAccountReport(String customerID) {
2386
2387         DataFactory dataFactory = DataFactory.INSTANCE;
2388         AccountReport accountReport =
2389             (AccountReport)dataFactory.create(AccountReport.class);
2390         List accountSummaries = accountReport.getAccountSummaries();
2391
2392         CheckingAccount checkingAccount =
2393             accountDataService.getCheckingAccount(customerID);
2394         AccountSummary checkingAccountSummary =
2395             (AccountSummary)dataFactory.create(AccountSummary.class);
2396
2397         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());
2398     };
2399
2400     checkingAccountSummary.setAccountType("checking");
2401     checkingAccountSummary.setBalance(fromUSDollarToCurrency
2402         (checkingAccount.getBalance()));
2403
2404     accountSummaries.add(checkingAccountSummary);
2405
2406     SavingsAccount savingsAccount =
2407         accountDataService.getSavingsAccount(customerID);
2408     AccountSummary savingsAccountSummary =
2409         (AccountSummary)dataFactory.create(AccountSummary.class);
2410
2411     savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
2412     savingsAccountSummary.setAccountType("savings");
2413     savingsAccountSummary.setBalance(fromUSDollarToCurrency
2414         (savingsAccount.getBalance()));
2415     accountSummaries.add(savingsAccountSummary);
2416
2417     StockAccount stockAccount =
2418     accountDataService.getStockAccount(customerID);
2419     AccountSummary stockAccountSummary =
2420         (AccountSummary)dataFactory.create(AccountSummary.class);
2421     stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
2422     stockAccountSummary.setAccountType("stock");
2423     float balance= (stockQuoteService.getQuote(stockAccount.getSymbol())) *
2424         stockAccount.getQuantity();
2425     stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
2426     accountSummaries.add(stockAccountSummary);

```



```
2425
2426     return accountReport;
2427 }
2428
2429 @PermitAll
2430 public float fromUSDollarToCurrency(float value) {
2431
2432     if (currency.equals("USD")) return value; else
2433     if (currency.equals("EURO")) return value * 0.8f; else
2434     return 0.0f;
2435 }
2436 }
```

2437 Example 3. Usage of annotated security implementation policy for the java language.

2438 In this example, the implementation class as a whole is marked:

- 2439 • @RolesAllowed("customers") - indicating that customers have access to the  
2440 implementation as a whole
- 2441 • @RunAs("accountants") – indicating that the code in the implementation runs with the  
2442 permissions of accountants

2443 The getAccountReport(..) method is marked with @RolesAllowed({"customers", "accountants"}),  
2444 which indicates that this method can be called by both customers and accountants.

2445 The fromUSDollarToCurrency() method is marked with @PermitAll, which means that this method  
2446 can be called by any role.

2447

---

2448 **A. Acknowledgements**

2449 The following individuals have participated in the creation of this specification and are gratefully  
2450 acknowledged:

2451 **Participants:**

2452 [Participant Name, Affiliation | Individual Member]

2453 [Participant Name, Affiliation | Individual Member]

2454

---

## B. Non-Normative Text

2456

## C. Revision History

2457 [optional; should not be included in OASIS Standards]

2458

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

2459

2460