



Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Committee Draft 02, Revision 01 **+AnnotationsMerge**

03 February 2009

Specification URIs:

This Version:

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

Previous Version:

Latest Version:

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

Latest Approved Version:

Technical Committee:

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

Chair(s):

Simon Nash,	IBM
Michael Rowley,	BEA Systems
Mark 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 can be used by Java-based SCA specifications.

Deleted: may

Specifically, this specification covers:

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

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

Deleted: may

Status:

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

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

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

Deleted: may

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

Notices

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

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

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

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

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

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

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

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

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

Table of Contents

1	Introduction.....	6
1.1	Terminology.....	6
1.2	Normative References.....	6
1.3	Non-Normative References.....	7
2	Implementation Metadata.....	8
2.1	Service Metadata.....	8
2.1.1	@Service.....	8
2.1.2	Java Semantics of a Remotable Service.....	8
2.1.3	Java Semantics of a Local Service.....	8
2.1.4	@Reference.....	9
2.1.5	@Property.....	9
2.2	Implementation Scopes: @Scope, @Init, @Destroy.....	9
2.2.1	Stateless scope.....	9
2.2.2	Composite scope.....	10
3	Interface.....	11
3.1	Java interface element – <interface.java>.....	11
3.2	@Remotable.....	12
3.3	@Callback.....	12
4	Client API.....	13
4.1	Accessing Services from an SCA Component.....	13
4.1.1	Using the Component Context API.....	13
4.2	Accessing Services from non-SCA component implementations.....	13
4.2.1	ComponentContext.....	13
5	Error Handling.....	14
6	Asynchronous Programming.....	15
6.1	@OneWay.....	15
6.2	Callbacks.....	15
6.2.1	Using Callbacks.....	15
6.2.2	Callback Instance Management.....	17
6.2.3	Implementing Multiple Bidirectional Interfaces.....	17
6.2.4	Accessing Callbacks.....	18
7	Policy Annotations for Java.....	19
7.1	General Intent Annotations.....	19
7.2	Specific Intent Annotations.....	21
7.2.1	How to Create Specific Intent Annotations.....	21
7.3	Application of Intent Annotations.....	22
7.3.1	Inheritance And Annotation.....	22
7.4	Relationship of Declarative And Annotated Intents.....	24
7.5	Policy Set Annotations.....	24
7.6	Security Policy Annotations.....	25
7.6.1	Security Interaction Policy.....	25
7.6.2	Security Implementation Policy.....	26
8	Java API.....	29

8.1 Component Context.....	29
8.2 Request Context.....	30
8.3 ServiceReference.....	31
8.4 ServiceRuntimeException.....	31
8.5 ServiceUnavailableException.....	32
8.6 InvalidServiceException.....	32
8.7 Constants Interface.....	32
9 Java Annotations.....	33
9.1 @AllowsPassByReference.....	33
9.2 @Authentication.....	34
9.3 @Callback.....	34
9.4 @ComponentName.....	35
9.5 @Confidentiality.....	36
9.6 @Constructor.....	37
9.7 @Context.....	37
9.8 @Destroy.....	38
9.9 @EagerInit.....	39
9.10 @Init.....	39
9.11 @Integrity.....	40
9.12 @Intent.....	40
9.13 @OneWay.....	41
9.14 @PolicySet.....	42
9.15 @Property.....	42
9.16 @Qualifier.....	44
9.17 @Reference.....	44
9.17.1 ReInjection.....	47
9.18 @Remotable.....	48
9.19 @Requires.....	49
9.20 @Scope.....	50
9.21 @Service.....	51
10 WSDL to Java and Java to WSDL.....	53
10.1 JAX-WS Client Asynchronous API for a Synchronous Service.....	53
A. XML Schema: sca-interface-java.xsd.....	55
B. Conformance Items.....	56
C. Acknowledgements.....	57
D. Non-Normative Text.....	58
E. Revision History.....	59

1 Introduction

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

Deleted: may

Specifically, this specification covers:

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

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

Deleted: may

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

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

1.1 Terminology

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

1.2 Normative References

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

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

51 **1.3 Non-Normative References**

52 **None** None

53 2 Implementation Metadata

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

56 2.1 Service Metadata

57 2.1.1 @Service

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

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

66 2.1.2 Java Semantics of a Remotable Service

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

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

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

78 2.1.3 Java Semantics of a Local Service

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

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

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

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

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

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

94 2.1.4 @Reference

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

98 2.1.5 @Property

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

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

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

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

109 This document defines two scopes:

- 110 • STATELESS
- 111 • COMPOSITE

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

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

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

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

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

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

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

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

137 2.2.1 Stateless scope

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

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

146 2.2.2 Composite scope

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

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

Deleted: may

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

157 3 Interface

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

159 3.1 Java interface element – <interface.java>

160 The Java interface element is used in SCDL files in places where an interface is declared in terms
161 of a Java interface class. The Java interface element identifies the Java interface class and
162 optionally identifies a callback interface, where the first Java interface represents the forward
163 (service) call interface and the second interface represents the interface used to call back from the
164 service to the client.

165

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

167

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

169

170 The interface.java element has the following attributes:

- 171 • **interface (1..1)** – the Java interface class to use for the service interface. @interface MUST
172 be the fully qualified name of the Java interface class [JCA30001]
- 173 • **callbackInterface (0..1)** – the Java interface class to use for the callback interface.
174 @callbackInterface MUST be the fully qualified name of a Java interface used for callbacks
175 [JCA30002]

176

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

178

```
179 <interface.java interface="services.stockquote.StockQuoteService"  
180     callbackInterface="services.stockquote.StockQuoteServiceCallback" />
```

181

182 Here, the Java interface is defined in the Java class file
183 ./services/stockquote/StockQuoteService.class, where the root directory is defined by the
184 contribution in which the interface exists. Similarly, the callback interface is defined in the Java
185 class file ./services/stockquote/StockQuoteServiceCallback.class.

186 Note that the Java interface class identified by the @interface attribute can contain a Java
187 @Callback annotation which identifies a callback interface. If this is the case, then it is not
188 necessary to provide the @callbackInterface attribute. However, if the Java interface class
189 identified by the @interface attribute does contain a Java @Callback annotation, then the Java
190 interface class identified by the @callbackInterface attribute MUST be the same interface class.
191 [JCA30003]

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

196

197

198 3.2 @Remotable

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

203 3.3 @Callback

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

208 4 Client API

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

Deleted: may

211 4.1 Accessing Services from an SCA Component

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

Deleted: may

216 4.1.1 Using the Component Context API

217 | When a component implementation needs access to a service where the reference to the service is
218 | not known at compile time, the reference can be located using the component's
219 | ComponentContext.

220 4.2 Accessing Services from non-SCA component implementations

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

223 4.2.1 ComponentContext

224 | Non-SCA client code can use the ComponentContext API to perform operations against a
225 | component in an SCA domain. How client code obtains a reference to a ComponentContext is
226 | runtime specific.

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

228

```
229 | ComponentContext context = // obtained via host environment-specific means  
230 | HelloService helloService =  
231 |     context.getService(HelloService.class, "HelloService");  
232 | String result = helloService.hello("Hello World!");
```

233 5 Error Handling

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

Deleted: may

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

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

240 6 Asynchronous Programming

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

- 248 • support for non-blocking method calls
- 249 • callbacks

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

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 a
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 Callbacks

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

- 267 • an interface for the provided service
- 268 • a callback interface that must be provided by the client

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

Deleted: may

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

Deleted: may

275 6.2.1 Using Callbacks

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

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

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

```
287 package somepackage;
288 import org.osoa.sca.annotation.Callback;
289 import org.osoa.sca.annotation.Remotable;
290 @Remotable
291 @Callback(QuotationCallback.class)
292 public interface Quotation {
293     double requestQuotation(String productCode, int quantity);
294 }
295
296 @Remotable
297 public interface QuotationCallback {
298     String getState();
299     String getZipCode();
300     String getCreditRating();
301 }
302
```

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

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

```
312 @Callback
313 protected QuotationCallback callback;
314
315 public double requestQuotation(String productCode, int quantity) {
316     double price = getPrice(productCode, quantity);
317     double discount = 0;
318     if (quantity > 1000 && callback.getState().equals("FL")) {
319         discount = 0.05;
320     }
321     if (quantity > 10000 && callback.getCreditRating().charAt(0) == 'A') {
322         discount += 0.05;
323     }
324     return price * (1-discount);
325 }
326 }
327
```

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

```
331
332 public class ClientImpl implements ClientService, QuotationCallback {
333
334     private QuotationService myService;
335
336     @Reference
337     public void setMyService(QuotationService service) {
338         myService = service;
339     }
340 }

```

```

340
341     public void aClientMethod() {
342         ...
343         double quote = myService.requestQuotation("AB123", 2000);
344         ...
345     }
346
347     public String getState() {
348         return "TX";
349     }
350     public String getZipCode() {
351         return "78746";
352     }
353     public String getCreditRating() {
354         return "AA";
355     }
356 }

```

357
358 In this example the callback is **stateless**, i.e., the callback requests do not need any information
359 relating to the original service request. For a callback that needs information relating to the
360 original service request (a **stateful** callback), this information can be passed to the client by the
361 service provider as parameters on the callback request..

362 6.2.2 Callback Instance Management

363 Instance management for callback requests received by the client of the bidirectional service is
364 handled in the same way as instance management for regular service requests. If the client
365 implementation has STATELESS scope, the callback is dispatched using a newly initialized
366 instance. If the client implementation has COMPOSITE scope, the callback is dispatched using the
367 same shared instance that is used to dispatch regular service requests.

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

374 6.2.3 Implementing Multiple Bidirectional Interfaces

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

```

381
382 @Callback
383 protected MyService1Callback callback1;
384
385 @Callback
386 protected MyService2Callback callback2;

```

387
388 If a single callback has a type that is compatible with multiple declared callback fields, then all of
389 them will be set.

390 6.2.4 Accessing Callbacks

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

394
395 | A reference implementing the callback service interface can be obtained using
396 `ServiceReference.getService()`.

Deleted: may

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

```
398 @Callback
399 protected ServiceReference<MyCallback> callback;
400
401 public void someMethod() {
402     MyCallback myCallback = callback.getCallback();    ...
403
404     myCallback.receiveResult(theResult);
405 }
406
407
408
```

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

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

Deleted: may

```
415 public void someMethod() {
416     MyCallback myCallback =
417         ComponentContext.getRequestContext().getCallback();
418     ...
419     myCallback.receiveResult(theResult);
420 }
421
422
423
424
```

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

```
427 @Context
428 protected RequestContext requestContext;
429
430 void receiveResult(Object theResult) {
431     Object refParams =
432         requestContext.getServiceReference().getCallbackID();
433     ...
434 }
435
```

436
437 This is necessary if the service implementation has **COMPOSITE** scope, because callback injection
438 is not performed for composite-scoped implementations.

439

7 Policy Annotations for Java

440 SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which
 441 influence how implementations, services and references behave at runtime. The policy facilities
 442 are described in the SCA Policy Framework specification [POLICY]. In particular, the facilities
 443 include Intents and Policy Sets, where intents express abstract, high-level policy requirements and
 444 policy sets express low-level detailed concrete policies.

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

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

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

462 The SCA Java Common Annotations specification supports using the Common Annotation for Java
 463 Platform specification (JSR-250) [JSR-250]. An implication of adopting the common annotation
 464 for Java platform specification is that the SCA Java specification support consistent annotation and
 465 Java class inheritance relationships.

466

7.1 General Intent Annotations

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

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

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

473 _____
 474 _____ *"{f" + Namespace URI + "}" + intentname*

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

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

```
480 public static final String SCA_PREFIX=  

  481 _____  

  482 public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";  

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

  484
```

485 [Notice that, by convention, qualified intents include the qualifier as part of the name of the](#)
486 [constant, separated by an underscore. These intent constants are defined in the file that defines](#)
487 [an annotation for the intent \(annotations for intents, and the formal definition of these constants,](#)
488 [are covered in a following section\).](#)

489 [Multiple intents \(qualified or not\) are expressed as separate strings within an array declaration.](#)

490 [An example of the @Requires annotation with 2 qualified intents \(from the Security domain\)](#)
491 [follows:](#)

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

493

494 [This attaches the intents "confidentiality.message" and "integrity.message".](#)

495 [The following is an example of a reference requiring support for confidentiality:](#)

```
496     package com.foo;
497
498     import static org.oasisopen.sca.annotation.Confidentiality.*;
499     import static org.oasisopen.sca.annotation.Reference;
500
501     public class Foo {
502         @Requires(CONFIDENTIALITY)
503         @Reference
504         public void setBar(Bar bar) {
505             ...
506         }
507     }
508
```

509 [Users can also choose to only use constants for the namespace part of the QName, so that they](#)
510 [can add new intents without having to define new constants. In that case, this definition would](#)
511 [instead look like this:](#)

```
512     package com.foo;
513
514     import static org.oasisopen.sca.Constants.*;
515     import static org.oasisopen.sca.annotation.Reference;
516     import static org.oasisopen.sca.annotation.Requires;
517
518     public class Foo {
519         @Requires(SCA_PREFIX+"confidentiality")
520         @Reference
521         public void setBar(Bar bar) {
522             ...
523         }
524     }
525
```

526 [The formal syntax for the @Requires annotation follows:](#)

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

528 [where](#)

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

530

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

532

7.2 Specific Intent Annotations

Formatted: Bullets and Numbering

533

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

537

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

538

539

540

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

541

542

543

```
@Integrity
```

544

545

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

546

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

547

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

548

549

550

The general form of specific intent annotations is:

551

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

552

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

553

```
Intent ::= NCName
```

554

```
qualifiers ::= "qualifier" (, "qualifier")*
```

555

```
qualifier ::= NCName(.qualifier)?
```

556

Formatted: French France

Formatted: French France

557

7.2.1 How to Create Specific Intent Annotations

Formatted: Bullets and Numbering

558

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

559

560

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

561

562

563

564

565

566

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

567

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

568

569

See section @Intent for the formal definition of the @Intent annotation.

570

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

571

572

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

573

574

575

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

576

577

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

578

579 [See section @Qualifier for the formal definition of the @Qualifier annotation.](#)

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

582 **7.3 Application of Intent Annotations**

Formatted: Bullets and Numbering

583 [The SCA Intent annotations can be applied to the following Java elements:](#)

- 584 [• Java class](#)
- 585 [• Java interface](#)
- 586 [• Method](#)
- 587 [• Field](#)
- 588 [• Constructor parameter](#)

Formatted: Bullets and Numbering

589 [Where multiple intent annotations \(general or specific\) are applied to the same Java element, they](#)
590 [are additive in effect. An example of multiple policy annotations being used together follows:](#)

```
591 @Authentication  
592 @Requires\({CONFIDENTIALITY\_MESSAGE, INTEGRITY\_MESSAGE}\)
```

593 [In this case, the effective intents are "authentication", "confidentiality.message" and](#)
594 ["integrity.message".](#)

595 [If an annotation is specified at both the class/interface level and the method or field level, then](#)
596 [the method or field level annotation completely overrides the class level annotation of the same](#)
597 [base intent name.](#)

598 [The intent annotation can be applied either to classes or to class methods when adding annotated](#)
599 [policy on SCA services. Applying an intent to the setter method in a reference injection approach](#)
600 [allows intents to be defined at references.](#)

601 **7.3.1 Inheritance And Annotation**

Formatted: Bullets and Numbering

602 [The inheritance rules for annotations are consistent with the common annotation specification, JSR](#)
603 [250.](#)

604 [The following example shows the inheritance relations of intents on classes, operations, and super](#)
605 [classes.](#)

```
606 package services.hello;  
607 import org.oasisopen.sca.annotation.Remotable;  
608 import org.oasisopen.sca.annotation.Integrity;  
609 import org.oasisopen.sca.annotation.Authentication;
```

```
610  
611 @Integrity\("transport"\)  
612 @Authentication  
613 public class HelloService {  
614     @Integrity  
615     @Authentication\("message"\)  
616     public String hello(String message) {...}  
617  
618     @Integrity  
619     @Authentication\("transport"\)  
620     public String helloThere() {...}  
621 }  
622
```

```
623 package services.hello;  
624 import org.oasisopen.sca.annotation.Remotable;  
625 import org.oasisopen.sca.annotation.Confidentiality;  
626 import org.oasisopen.sca.annotation.Authentication;
```

```

627
628     @Confidentiality("message")
629     public class HelloChildService extends HelloService {
630         @Confidentiality("transport")
631         public String hello(String message) {...}
632         @Authentication
633         String helloWorld() {...}
634     }

```

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

636
637 The effective intent annotation on the helloWorld method is Integrity("transport"),
638 @Authentication, and @Confidentiality("message").

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

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

643 The effective intent annotation on the hello method of the HelloService is @Integrity and
644 @Authentication("message").

645
646 The listing below contains the equivalent declarative security interaction policy of the HelloService
647 and HelloChildService implementation corresponding to the Java interfaces and classes shown in
648 Example 2a.

```

649
650     <?xml version="1.0" encoding="ASCII"?>
651     <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
652             name="HelloServiceComposite" >
653         <service name="HelloService" requires="integrity/transport
654             authentication">
655             ...
656         </service>
657         <service name="HelloChildService" requires="integrity/transport
658             authentication confidentiality/message">
659             ...
660         </service>
661         ...
662
663         <component name="HelloServiceComponent">*
664             <implementation.java class="services.hello.HelloService"/>
665             <operation name="hello" requires="integrity
666                 authentication/message"/>
667             <operation name="helloThere"
668                 requires="integrity
669                 authentication/transport"/>
670         </component>
671         <component name="HelloChildServiceComponent">*
672             <implementation.java
673                 class="services.hello.HelloChildService" />
674             <operation name="hello"
675                 requires="confidentiality/transport"/>
676             <operation name="helloThere" requires=" integrity/transport
677                 authentication"/>
678             <operation name="helloWorld" requires="authentication"/>
679         </component>
680

```

681 ...
682
683 </composite>

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

686

687 **7.4 Relationship of Declarative And Annotated Intents**

Formatted: Bullets and Numbering

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

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

695 **7.5 Policy Set Annotations**

Formatted: Bullets and Numbering

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

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

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

705

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

707 An example of the @PolicySets annotation:

708

```
709       @Reference(name="helloService", required=true)  
710       @PolicySets({ MY_NS + "WS_Encryption_Policy",  
711                   MY_NS + "WS_Authentication_Policy" })  
712       public setHelloService>HelloService service) {  
713               ...  
714       }
```

715

716 In this case, the Policy Sets WS Encryption Policy and WS Authentication Policy are applied, both
717 using the namespace defined for the constant MY_NS.

718 PolicySets must satisfy intents expressed for the implementation when both are present, according
719 to the rules defined in the Policy Framework specification [POLICY].

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

- 721 • Java class
- 722 • Java interface
- 723 • Method
- 724 • Field
- 725 • Constructor parameter

Formatted: Bullets and Numbering

7.6 Security Policy Annotations

This section introduces annotations for SCA's security intents, as defined in the SCA Policy Framework specification [POLICY].

7.6.1 Security Interaction Policy

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

- `@Integrity`
- `@Confidentiality`
- `@Authentication`

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

- `message`
- `transport`

The formal definitions of the `@Authentication`, `@Confidentiality` and `@Integrity` annotations are found in the sections `@Authentication`, `@Confidentiality` and `@Integrity`.

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

```
package services.hello;
//Interface for HelloService
public interface HelloService {
    String hello(String helloMsg);
}

package services.client;
// Interface for ClientService
public interface ClientService {
    public void clientMethod();
}

// Implementation class for ClientService
package services.client;

import services.hello.HelloService;
import org.oasisopen.sca.annotation.*;

@Service(ClientService.class)
public class ClientServiceImpl implements ClientService {
    private HelloService helloService;

    @Reference(name="helloService", required=true)
    @Integrity("message")
    @Authentication("message")
    public void setHelloService(HelloService service) {
        helloService = service;
    }

    public void clientMethod() {
        String result = helloService.hello("Hello World!");
    }
}
```

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

```
776     ...  
777     }  
778 }  
779
```

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

781 **7.6.2 Security Implementation Policy**

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

- 785 • RunAs

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

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

- 791 • RolesAllowed

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

- 798 • PermitAll

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

- 801 • DenyAll

802 No parameters. When present, denies access to all roles.

- 804 • DeclareRoles

805 Takes as a parameter a string or an array of strings which identify one or more role names
806 that form the set of roles used by the implementation.
807 eg. `@DeclareRoles({"Manager", "Employee", "Customer"})`

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

809 For a full explanation of these intents, see the Policy Framework specification [POLICY].

810 **7.6.2.1 Annotated Implementation Policy Example**

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

```
812 package services.account;  
813 @Remotable  
814 public interface AccountService {  
815     AccountReport getAccountReport(String customerID);  
816 }  
817
```

818
819 The following is a full listing of the `AccountServiceImpl` class, showing the Service it implements,
820 plus the service references it makes and the settable properties that it has, along with a set of
821 implementation policy annotations:

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

```

823 package services.account;
824 import java.util.List;
825 import commonj.sdo.DataFactory;
826 import org.oasisopen.sca.annotation.Property;
827 import org.oasisopen.sca.annotation.Reference;
828 import org.oasisopen.sca.annotation.RolesAllowed;
829 import org.oasisopen.sca.annotation.RunAs;
830 import org.oasisopen.sca.annotation.PermitAll;
831 import services.accountdata.AccountDataService;
832 import services.accountdata.CheckingAccount;
833 import services.accountdata.SavingsAccount;
834 import services.accountdata.StockAccount;
835 import services.stockquote.StockQuoteService;
836 @RolesAllowed("customers")
837 @RunAs("accountants" )
838 public class AccountServiceImpl implements AccountService {
839
840     @Property
841     protected String currency = "USD";
842
843     @Reference
844     protected AccountDataService accountDataService;
845     @Reference
846     protected StockQuoteService stockQuoteService;
847
848     @RolesAllowed({"customers", "accountants"})
849     public AccountReport getAccountReport(String customerID) {
850
851         DataFactory dataFactory = DataFactory.INSTANCE;
852         AccountReport accountReport =
853             (AccountReport)dataFactory.create(AccountReport.class);
854         List accountSummaries = accountReport.getAccountSummaries();
855
856         CheckingAccount checkingAccount =
857             accountDataService.getCheckingAccount(customerID);
858         AccountSummary checkingAccountSummary =
859             (AccountSummary)dataFactory.create(AccountSummary.class);
860
861         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber(
862         ));
863         checkingAccountSummary.setAccountType("checking");
864         checkingAccountSummary.setBalance(fromUSDollarToCurrency
865             (checkingAccount.getBalance()));
866         accountSummaries.add(checkingAccountSummary);
867
868         SavingsAccount savingsAccount =
869             accountDataService.getSavingsAccount(customerID);
870         AccountSummary savingsAccountSummary =
871             (AccountSummary)dataFactory.create(AccountSummary.class);
872
873         savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber(
874         ));
875         savingsAccountSummary.setAccountType("savings");
876         savingsAccountSummary.setBalance(fromUSDollarToCurrency
877             (savingsAccount.getBalance()));
878         accountSummaries.add(savingsAccountSummary);
879
880         StockAccount stockAccount =
881             accountDataService.getStockAccount(customerID);

```

```

881     AccountSummary stockAccountSummary =
882         (AccountSummary)dataFactory.create(AccountSummary.class);
883     stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
884     stockAccountSummary.setAccountType("stock");
885     float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*
886         stockAccount.getQuantity();
887     stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
888     accountSummaries.add(stockAccountSummary);
889
890     return accountReport;
891 }
892
893 @PermitAll
894 public float fromUSDollarToCurrency(float value) {
895
896     if (currency.equals("USD")) return value; else
897     if (currency.equals("EURO")) return value * 0.8f; else
898     return 0.0f;
899 }
900 }

```

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

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

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

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

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

← - - - - **Formatted:** Bullets and Numbering

Formatted: Bullets and Numbering

911 **8 Java API**

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

913 **8.1 Component Context**

Formatted: Bullets and Numbering

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

```
915
916 package org.oasisopen.sca;
917
918 public interface ComponentContext {
919     String getURI();
920
921     <B> B getService(Class<B> businessInterface, String referenceName);
922
923     <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
924                                             String referenceName);
925
926     <B> Collection<B> getServices(Class<B> businessInterface,
927                                String referenceName);
928
929     <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
930                                                           businessInterface, String referenceName);
931
932     <B> ServiceReference<B> createSelfReference(Class<B>
933                                               businessInterface);
934
935     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
936                                               String serviceName);
937
938     <B> B getProperty(Class<B> type, String propertyName);
939
940     <B, R extends ServiceReference<B>> R cast(B target)
941         throws IllegalArgumentException;
942
943     RequestContext getRequestContext();
944
945
946 }
```

- 947
- 948 • **getURI()** - returns the absolute URI of the component within the SCA domain
 - 949 • **getService(Class businessInterface, String referenceName)** – Returns a proxy for
950 the reference defined by the current component. The getService() method takes as its
951 input arguments the Java type used to represent the target service on the client and the
952 name of the service reference. It returns an object providing access to the service. The
953 returned object implements the Java interface the service is typed with. This method
954 MUST throw an IllegalArgumentException if the reference has multiplicity greater than
955 one.
 - 956 • **getServiceReference(Class businessInterface, String referenceName)** – Returns a
957 ServiceReference defined by the current component. This method MUST throw an
958 IllegalArgumentException if the reference has multiplicity greater than one.

- 959 • **getServices(Class businessInterface, String referenceName)** – Returns a list of
960 typed service proxies for a business interface type and a reference name.
- 961 • **getServiceReferences(Class businessInterface, String referenceName)** –Returns a
962 list typed service references for a business interface type and a reference name.
- 963 • **createSelfReference(Class businessInterface)** – Returns a ServiceReference that can
964 be used to invoke this component over the designated service.
- 965 • **createSelfReference(Class businessInterface, String serviceName)** – Returns a
966 ServiceReference that can be used to invoke this component over the designated service.
967 Service name explicitly declares the service name to invoke
- 968 • **getProperty (Class type, String propertyName)** - Returns the value of an SCA
969 property defined by this component.
- 970 • **getRequestContext()** - Returns the context for the current SCA service request, or null if
971 there is no current request or if the context is unavailable. This method MUST return non-
972 null when invoked during the execution of a Java business method for a service operation
973 or callback operation, on the same thread that the SCA runtime provided, and MUST
974 return null in all other cases.
- 975 • **cast(B target)** - Casts a type-safe reference to a ServiceReference

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

Deleted: may

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

```
981 private ComponentContext componentContext;
982
983 @Context
984 public void setContext(ComponentContext context) {
985     componentContext = context;
986 }
987
988 public void doSomething() {
989     HelloWorld service =
990     componentContext.getService(HelloWorld.class, "HelloWorldComponent");
991     service.hello("hello");
992 }
993
```

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

Formatted: Bullets and
Numbering

997 | **8.2 Request Context**

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

```
999
1000 package org.oasisopen.sca;
1001
1002 import javax.security.auth.Subject;
1003
1004 public interface RequestContext {
1005
1006     Subject getSecuritySubject();
1007
1008     String getServiceName();

```

```

1009     <CB> ServiceReference<CB> getCallbackReference();
1010     <CB> CB getCallback();
1011     <B> ServiceReference<B> getServiceReference();
1012
1013 }
1014

```

1015 The RequestContext interface has the following methods:

- 1016 • **getSecuritySubject()** – Returns the JAAS Subject of the current request
- 1017 • **getServiceName()** – Returns the name of the service on the Java implementation the
1018 request came in on
- 1019 • **getCallbackReference()** – Returns a service reference to the callback as specified by the
1020 caller. This method returns null when called for a service request whose interface is not
1021 bidirectional or when called for a callback request.
- 1022 • **getCallback()** – Returns a proxy for the callback as specified by the caller. Similar to the
1023 getCallbackReference() method, this method returns null when called for a service request
1024 whose interface is not bidirectional or when called for a callback request.
- 1025 • **getServiceReference()** – When invoked during the execution of a service operation, this
1026 method MUST return a ServiceReference that represents the service that was invoked.
1027 When invoked during the execution of a callback operation, this method MUST return a
1028 CallableReference that represents the callback that was invoked.

1029 **8.3 ServiceReference**

1030 ServiceReferences can be injected using the @Reference annotation on a field, a setter method,
1031 or constructor parameter taking the type ServiceReference. The detailed description of the usage
1032 of these methods is described in the section on Asynchronous Programming in this document.

1033 The following Java code defines the **ServiceReference** interface:

```

1034 package org.oasisopen.sca;
1035
1036 public interface ServiceReference<B> extends java.io.Serializable {
1037
1038     B getService();
1039     Class<B> getBusinessInterface();
1040 }
1041

```

1042 The ServiceReference interface has the following methods:

- 1044 • **getService()** - Returns a type-safe reference to the target of this reference. The instance
1045 returned is guaranteed to implement the business interface for this reference. The value
1046 returned is a proxy to the target that implements the business interface associated with this
1047 reference.
- 1048 • **getBusinessInterface()** – Returns the Java class for the business interface associated with
1049 this reference.

1050 **8.4 ServiceRuntimeException**

1051 The following snippet shows the **ServiceRuntimeException**.

```

1052
1053 package org.oasisopen.sca;
1054
1055 public class ServiceRuntimeException extends RuntimeException {

```

Formatted: Bullets and
Numbering

Deleted: may

Formatted: Bullets and
Numbering

1056 ...
1057 }
1058
1059

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

Formatted: Bullets and Numbering

1060 **8.5 ServiceUnavailableException**

1061 The following snippet shows the *ServiceUnavailableException*.

```
1062 package org.oasisopen.sca;  
1063  
1064 public class ServiceUnavailableException extends ServiceRuntimeException {  
1065     ...  
1066 }  
1067  
1068
```

1069 This exception signals problems in the interaction with remote services. These are exceptions
1070 that can be transient, so retrying is appropriate. Any exception that is a ServiceRuntimeException
1071 that is *not* a ServiceUnavailableException is unlikely to be resolved by retrying the operation, since
1072 it most likely requires human intervention

Deleted: may

1073 **8.6 InvalidServiceException**

1074 The following snippet shows the *InvalidServiceException*.

```
1075 package org.oasisopen.sca;  
1076  
1077 public class InvalidServiceException extends ServiceRuntimeException {  
1078     ...  
1079 }  
1080  
1081
```

1082 This exception signals that the ServiceReference is no longer valid. This can happen when the
1083 target of the reference is undeployed. This exception is not transient and therefore is unlikely to
1084 be resolved by retrying the operation and will most likely require human intervention.

Formatted: Bullets and Numbering

1085 **8.7 Constants Interface**

1086 The SCA *Constants* interface defines a number of constant values that are used in the SCA Java
1087 APIs and Annotations. The following snippet shows the Constants interface:

```
1088 package org.oasisopen.sca;  
1089  
1090 public interface Constants {  
1091     String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";  
1092     String SCA_PREFIX = "{"+SCA_NS+"}";  
1093 }  
1094
```

Formatted: Heading 2,H2

Formatted: Space Before: 0 pt, After: 0 pt

1095 9 Java Annotations

1096 This section provides definitions of all the Java annotations which apply to SCA.

1097 This specification places constraints on some annotations that are not detectable by a Java
1098 compiler. For example, the definition of the @Property and @Reference annotations indicate that
1099 they are allowed on parameters, but sections 8.14 and 8.15 constrain those definitions to
1100 constructor parameters. An SCA runtime MUST verify the proper use of all annotations and if an
1101 annotation is improperly used, the SCA runtime MUST NOT run the component which uses the
1102 invalid implementation code.

1103 SCA annotations are not allowed on static methods and static fields. It is an error to use an SCA
1104 annotation on a static method or a static field of an implementation class and the SCA runtime
1105 MUST NOT instantiate such an implementation class.

1106 9.1 @AllowsPassByReference

1107 The following Java code defines the **@AllowsPassByReference** annotation:

```
1108
1109 package org.oasisopen.sca.annotation;
1110
1111 import static java.lang.annotation.ElementType.TYPE;
1112 import static java.lang.annotation.ElementType.METHOD;
1113 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1114 import java.lang.annotation.Retention;
1115 import java.lang.annotation.Target;
1116
1117 @Target({TYPE, METHOD})
1118 @Retention(RUNTIME)
1119 public @interface AllowsPassByReference {
1120
1121 }
1122
```

1123 The **@AllowsPassByReference** annotation is used on implementations of remotable interfaces to
1124 indicate that interactions with the service from a client within the same address space are allowed
1125 to use pass by reference data exchange semantics. The implementation promises that its by-value
1126 semantics will be maintained even if the parameters and return values are actually passed by-
1127 reference. This means that the service will not modify any operation input parameter or return
1128 value, even after returning from the operation. Either a whole class implementing a remotable
1129 service or an individual remotable service method implementation can be annotated using the
1130 @AllowsPassByReference annotation.

1131 @AllowsPassByReference has no attributes

1132

1133 The following snippet shows a sample where @AllowsPassByReference is defined for the
1134 implementation of a service method on the Java component implementation class.

1135

```
1136 @AllowsPassByReference
1137 public String hello(String message) {
1138     ...
1139 }
```

1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192

9.2 @Authentication

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

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

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import static org.oasisopen.sca.Constants.SCA_PREFIX;

import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Inherited
@Target({TYPE, FIELD, METHOD, PARAMETER})
@Retention(RUNTIME)
@Intent(Authentication.AUTHENTICATION)
public @interface Authentication {
    String AUTHENTICATION = SCA_PREFIX + "authentication";
    String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";
    String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";

    /**
     * List of authentication qualifiers (such as "message"
     * or "transport").
     *
     * @return authentication qualifiers
     */
    @Qualifier
    String[] value() default "";
}
```

The **@Authentication** annotation is used to indicate that the invocation requires authentication. See the section on Application of Intent Annotations for samples and details.

9.3 @Callback

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

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

import static java.lang.annotation.ElementType.TYPE;
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(TYPE, METHOD, FIELD)
@Retention(RUNTIME)
public @interface Callback {
    Class<?> value() default Void.class;
}
```

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

1193 }

1194

1195

1196 The @Callback annotation is used to annotate a service interface with a callback interface, which
1197 takes the Java Class object of the callback interface as a parameter.

1198 The @Callback annotation has the following attribute:

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

1200

1201 | The @Callback annotation ~~can~~ also be used to annotate a method or a field of an SCA
1202 implementation class, in order to have a callback object injected

Deleted: may

1203

1204 The following snippet shows a @Callback annotation on an interface:

1205

```
1206 @Remotable  
1207 @Callback(MyServiceCallback.class)  
1208 public interface MyService {  
1209  
1210     void someAsyncMethod(String arg);  
1211 }  
1212
```

1213 An example use of the @Callback annotation to declare a callback interface follows:

1214

```
1215 package somepackage;  
1216 import org.oasisopen.sca.annotation.Callback;  
1217 import org.oasisopen.sca.annotation.Remotable;  
1218 @Remotable  
1219 @Callback(MyServiceCallback.class)  
1220 public interface MyService {  
1221  
1222     void someMethod(String arg);  
1223 }  
1224  
1225 @Remotable  
1226 public interface MyServiceCallback {  
1227  
1228     void receiveResult(String result);  
1229 }  
1230
```

1231 In this example, the implied component type is:

1232

```
1233 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >  
1234     <service name="MyService">  
1235         <interface.java interface="somepackage.MyService"  
1236             callbackInterface="somepackage.MyServiceCallback"/>  
1237     </service>  
1238 </componentType>  
1239
```

Formatted: Bullets and
Numbering

1240 **9.4 @ComponentName**

1241 The following Java code defines the @ComponentName annotation:

```

1242
1243 package org.oasisopen.sca.annotation;
1244
1245 import static java.lang.annotation.ElementType.METHOD;
1246 import static java.lang.annotation.ElementType.FIELD;
1247 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1248 import java.lang.annotation.Retention;
1249 import java.lang.annotation.Target;
1250
1251 @Target({METHOD, FIELD})
1252 @Retention(RUNTIME)
1253 public @interface ComponentName {
1254
1255 }
1256

```

1257 The @ComponentName annotation is used to denote a Java class field or setter method that is
1258 used to inject the component name.

1259 The following snippet shows a component name field definition sample.

```

1260
1261 @ComponentName
1262 private String componentName;
1263

```

1264 The following snippet shows a component name setter method sample.

```

1265
1266 @ComponentName
1267 public void setComponentName(String name) {
1268     //...
1269 }

```

Formatted: Bullets and
Numbering

1270 9.5 @Confidentiality

1271 The following Java code defines the **@Confidentiality** annotation:

```

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

```

```
1294 /**  
1295  * List of confidentiality qualifiers (such as "message" or  
1296  "transport").  
1297  *  
1298  * @return confidentiality qualifiers  
1299  */  
1300 @Qualifier  
1301 String[] value() default "";  
1302 }  
1303
```

1304 [The **@Confidentiality** annotation is used to indicate that the invocation requires confidentiality.](#)

1305 [See the section on Application of Intent Annotations for samples and details.](#)

Formatted: Bullets and
Numbering

1306 **9.6 @Constructor**

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

```
1308 package org.oasisopen.sca.annotation;  
1309  
1310 import static java.lang.annotation.ElementType.CONSTRUCTOR;  
1311 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1312 import java.lang.annotation.Retention;  
1313 import java.lang.annotation.Target;
```

```
1314 @Target(CONSTRUCTOR)  
1315 @Retention(RUNTIME)  
1316 public @interface Constructor { }  
1317  
1318  
1319
```

1320 The @Constructor annotation is used to mark a particular constructor to use when instantiating a
1321 Java component implementation. If this constructor has parameters, each of these parameters
1322 MUST have either a @Property annotation or a @Reference annotation.

1323 The following snippet shows a sample for the @Constructor annotation.

```
1324  
1325 public class HelloServiceImpl implements HelloService {  
1326  
1327     public HelloServiceImpl(){  
1328         ...  
1329     }  
1330  
1331     @Constructor  
1332     public HelloServiceImpl(@Property(name="someProperty") String  
1333     someProperty ){  
1334         ...  
1335     }  
1336  
1337     public String hello(String message) {  
1338         ...  
1339     }  
1340 }
```

1341 **9.7 @Context**

1342 The following Java code defines the **@Context** annotation:

Formatted: Bullets and
Numbering

```

1344 package org.oasisopen.sca.annotation;
1345
1346 import static java.lang.annotation.ElementType.METHOD;
1347 import static java.lang.annotation.ElementType.FIELD;
1348 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1349 import java.lang.annotation.Retention;
1350 import java.lang.annotation.Target;
1351
1352 @Target({METHOD, FIELD})
1353 @Retention(RUNTIME)
1354 public @interface Context {
1355
1356 }
1357

```

1358 The @Context annotation is used to denote a Java class field or a setter method that is used to
1359 inject a composite context for the component. The type of context to be injected is defined by the
1360 type of the Java class field or type of the setter method input argument; the type is either
1361 **ComponentContext** or **RequestContext**.

1362 The @Context annotation has no attributes.

1363

1364 The following snippet shows a ComponentContext field definition sample.

1365

```

1366 @Context
1367 protected ComponentContext context;
1368

```

1369 The following snippet shows a RequestContext field definition sample.

1370

```

1371 @Context
1372 protected RequestContext context;

```

1373 **9.8 @Destroy**

1374 The following Java code defines the **@Destroy** annotation:

1375

```

1376 package org.oasisopen.sca.annotation;
1377
1378 import static java.lang.annotation.ElementType.METHOD;
1379 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1380 import java.lang.annotation.Retention;
1381 import java.lang.annotation.Target;
1382
1383 @Target(METHOD)
1384 @Retention(RUNTIME)
1385 public @interface Destroy {
1386
1387 }
1388

```

1389 The @Destroy annotation is used to denote a single Java class method that will be called when the
1390 scope defined for the implementation class ends. The method MAY have any access modifier and
1391 MUST have a void return type and no arguments.

1392 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1393 when the scope defined for the implementation class ends. If the implementation class has a

1394 method with an @Destroy annotation that does not match these criteria, the SCA runtime MUST
1395 NOT instantiate the implementation class.

1396 The following snippet shows a sample for a destroy method definition.

1397

```
1398 @Destroy  
1399 public void myDestroyMethod() {  
1400     ...  
1401 }
```

Formatted: Bullets and Numbering

1402 **9.9 @EagerInit**

1403 The following Java code defines the **@EagerInit** annotation:

1404

```
1405 package org.oasisopen.sca.annotation;  
1406  
1407 import static java.lang.annotation.ElementType.TYPE;  
1408 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1409 import java.lang.annotation.Retention;  
1410 import java.lang.annotation.Target;  
1411  
1412 @Target (TYPE)  
1413 @Retention(RUNTIME)  
1414 public @interface EagerInit {  
1415  
1416 }  
1417
```

1418 The **@EagerInit** annotation is used to annotate the Java class of a COMPOSITE scoped
1419 implementation for eager initialization. When marked for eager initialization, the composite scoped
1420 instance is created when its containing component is started.

Formatted: Bullets and Numbering

1421 **9.10 @Init**

1422 The following Java code defines the **@Init** annotation:

1423

```
1424 package org.oasisopen.sca.annotation;  
1425  
1426 import static java.lang.annotation.ElementType.METHOD;  
1427 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1428 import java.lang.annotation.Retention;  
1429 import java.lang.annotation.Target;  
1430  
1431 @Target (METHOD)  
1432 @Retention(RUNTIME)  
1433 public @interface Init {  
1434  
1435 }  
1436  
1437
```

1438 The @Init annotation is used to denote a single Java class method that is called when the scope
1439 defined for the implementation class starts. The method MAY have any access modifier and MUST
1440 have a void return type and no arguments.

1441 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1442 after all property and reference injection is complete. If the implementation class has a method

1443 with an @Init annotation that does not match these criteria, the SCA runtime MUST NOT
1444 instantiate the implementation class.

1445 The following snippet shows an example of an init method definition.

1446

```
1447 @Init  
1448 public void myInitMethod() {  
1449     ...  
1450 }
```

Formatted: Bullets and
Numbering

1451 **9.11 @Integrity**

1452 The following Java code defines the **@Integrity** annotation:

1453

```
1454 package org.oasisopen.sca.annotation;
```

1455

```
1456 import static java.lang.annotation.ElementType.FIELD;  
1457 import static java.lang.annotation.ElementType.METHOD;  
1458 import static java.lang.annotation.ElementType.PARAMETER;  
1459 import static java.lang.annotation.ElementType.TYPE;  
1460 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1461 import static org.oasisopen.Constants.SCA_PREFIX;
```

1462

```
1463 import java.lang.annotation.Inherited;  
1464 import java.lang.annotation.Retention;  
1465 import java.lang.annotation.Target;
```

1466

```
1467 @Inherited
```

```
1468 @Target({TYPE, FIELD, METHOD, PARAMETER})
```

1469

```
1469 @Retention(RUNTIME)
```

1470

```
1470 @Intent(Integrity.INTEGRITY)
```

1471

```
1471 public @interface Integrity {  
1472     String INTEGRITY = SCA_PREFIX + "integrity";  
1473     String INTEGRITY_MESSAGE = INTEGRITY + ".message";  
1474     String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";
```

1475

```
1475     /**
```

```
1476     * List of integrity qualifiers (such as "message" or "transport").
```

1477

```
1477     * @return integrity qualifiers
```

1478

```
1478     */
```

1479

```
1479     @Qualifier  
1480     String[] value() default "";
```

1481

```
1481 }  
1482
```

1483

1484 The **@Integrity** annotation is used to indicate that the invocation requires integrity (ie no

1485 tampering of the messages between client and service).

1486

1487 See the section on Application of Intent Annotations for samples and details.

Formatted: Bullets and
Numbering

1488 **9.12 @Intent**

1489 The following Java code defines the **@Intent** annotation:

1490

```
1490 package org.osoa.sca.annotation;
```

1491

```
1491 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
```

1492

1493

```

1494 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1495 import java.lang.annotation.Retention;
1496 import java.lang.annotation.Target;
1497
1498 @Target({ANNOTATION_TYPE})
1499 @Retention(RUNTIME)
1500 public @interface Intent {
1501     /**
1502      * The qualified name of the intent, in the form defined by
1503      * {@link javax.xml.namespace.QName#toString}.
1504      * @return the qualified name of the intent
1505      */
1506     String value() default "";
1507
1508     /**
1509      * The XML namespace for the intent.
1510      * @return the XML namespace for the intent
1511      */
1512     String targetNamespace() default "";
1513
1514     /**
1515      * The name of the intent within its namespace.
1516      * @return name of the intent within its namespace
1517      */
1518     String localPart() default "";
1519 }
1520

```

1521 The @Intent annotation is used for the creation of new annotations for specific intents. It is not
 1522 expected that the @Intent annotation will be used in application code.

1523 See the section "How to Create Specific Intent Annotations" for details and samples of how to
 1524 define new intent annotations.

Formatted: Bullets and
Numbering

1525 **9.13 @OneWay**

1526 The following Java code defines the **@OneWay** annotation:

```

1527
1528 package org.oasisopen.sca.annotation;
1529
1530 import static java.lang.annotation.ElementType.METHOD;
1531 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1532 import java.lang.annotation.Retention;
1533 import java.lang.annotation.Target;
1534
1535 @Target(METHOD)
1536 @Retention(RUNTIME)
1537 public @interface OneWay {
1538
1539
1540 }
1541

```

1542 The @OneWay annotation is used on a Java interface or class method to indicate that invocations
 1543 will be dispatched in a non-blocking fashion as described in the section on Asynchronous
 1544 Programming.

1545 The @OneWay annotation has no attributes.

1546 The following snippet shows the use of the @OneWay annotation on an interface.

```
1547 package services.hello;
1548
1549 import org.oasisopen.sca.annotation.OneWay;
1550
1551 public interface HelloService {
1552     @OneWay
1553     void hello(String name);
1554 }
```

9.14 @PolicySet

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

```
1555 package org.oasisopen.sca.annotation;
1556
1557 import static java.lang.annotation.ElementType.FIELD;
1558 import static java.lang.annotation.ElementType.METHOD;
1559 import static java.lang.annotation.ElementType.PARAMETER;
1560 import static java.lang.annotation.ElementType.TYPE;
1561 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1562
1563 import java.lang.annotation.Retention;
1564 import java.lang.annotation.Target;
1565
1566 @Target({TYPE, FIELD, METHOD, PARAMETER})
1567 @Retention(RUNTIME)
1568 public @interface PolicySets {
1569     /**
1570      * Returns the policy sets to be applied.
1571      *
1572      * @return the policy sets to be applied
1573      */
1574     String[] value() default "";
1575 }
1576
1577
1578
1579
```

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

See the section "Policy Set Annotations" for details and samples.

9.15 @Property

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

```
1584 package org.oasisopen.sca.annotation;
1585
1586 import static java.lang.annotation.ElementType.METHOD;
1587 import static java.lang.annotation.ElementType.FIELD;
1588 import static java.lang.annotation.ElementType.PARAMETER;
1589 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1590
1591 import java.lang.annotation.Retention;
1592 import java.lang.annotation.Target;
1593
1594 @Target({METHOD, FIELD, PARAMETER})
1595 @Retention(RUNTIME)
1596 public @interface Property {
1597     String name() default "";
1598 }
```

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

```
1599     boolean required() default true;
1600 }
1601
```

1602 The @Property annotation is used to denote a Java class field, a setter method, or a constructor
1603 parameter that is used to inject an SCA property value. The type of the property injected, which
1604 can be a simple Java type or a complex Java type, is defined by the type of the Java class field or
1605 the type of the input parameter of the setter method or constructor.

1606 | The @Property annotation can be used on fields, on setter methods or on a constructor method
1607 parameter. However, the @Property annotation MUST NOT be used on a class field that is declared
1608 as final. Deleted: may

1609 | Properties can also be injected via setter methods even when the @Property annotation is not
1610 present. However, the @Property annotation must be used in order to inject a property onto a
1611 non-public field. In the case where there is no @Property annotation, the name of the property is
1612 the same as the name of the field or setter. Deleted: may

1613 Where there is both a setter method and a field for a property, the setter method is used.

1614 The @Property annotation has the following attributes:

- 1615 • **name (optional)** – the name of the property. For a field annotation, the default is the
1616 name of the field of the Java class. For a setter method annotation, the default is the
1617 JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a
1618 constructor parameter annotation, there is no default and the name attribute MUST be
1619 present.
- 1620 • **required (optional)** – specifies whether injection is required, defaults to true. For a
1621 constructor parameter annotation, this attribute MUST have the value true.

1622

1623 The following snippet shows a property field definition sample.

1624

```
1625 @Property(name="currency", required=true)
1626 protected String currency;
```

1627

1628 The following snippet shows a property setter sample

1629

```
1630 @Property(name="currency", required=true)
1631 public void setCurrency( String theCurrency ) {
1632     ....
1633 }
```

1634

1635 If the property is defined as an array or as any type that extends or implements
1636 **java.util.Collection**, then the implied component type has a property with a **many** attribute set to
1637 true.

1638 The following snippet shows the definition of a configuration property using the @Property
1639 annotation for a collection.

```
1640 ...
1641 private List<String> helloConfigurationProperty;
1642
1643 @Property(required=true)
1644 public void setHelloConfigurationProperty(List<String> property) {
1645     helloConfigurationProperty = property;
1646 }
```

1647 ...

Formatted: Bullets and Numbering

1648 9.16 @Qualifier

1649 The following Java code defines the **@Qualifier** annotation:

```
1650 package org.oasisopen.sca.annotation;  
1651  
1652 import static java.lang.annotation.ElementType.METHOD;  
1653 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1654  
1655 import java.lang.annotation.Retention;  
1656 import java.lang.annotation.Target;  
1657  
1658 @Target(METHOD)  
1659 @Retention(RUNTIME)  
1660 public @interface Qualifier {  
1661 }  
1662  
1663
```

1664 The @Qualifier annotation is applied to an attribute of an intent annotation definition, defined
1665 using the @Intent annotation, to indicate that the attribute provides qualifiers for the intent. The
1666 @Qualifier annotation MUST be used in an intent annotation definition where the intent has
1667 qualifiers.

1668 See the section "How to Create Specific Intent Annotations" for details and samples of how to
1669 define new intent annotations.

1670 9.17 @Reference

Formatted: Bullets and Numbering

1671 The following Java code defines the **@Reference** annotation:

```
1672  
1673 package org.oasisopen.sca.annotation;  
1674  
1675 import static java.lang.annotation.ElementType.METHOD;  
1676 import static java.lang.annotation.ElementType.FIELD;  
1677 import static java.lang.annotation.ElementType.PARAMETER;  
1678 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1679 import java.lang.annotation.Retention;  
1680 import java.lang.annotation.Target;  
1681 @Target({METHOD, FIELD, PARAMETER})  
1682 @Retention(RUNTIME)  
1683 public @interface Reference {  
1684  
1685     String name() default "";  
1686     boolean required() default true;  
1687 }  
1688
```

1689 The @Reference annotation type is used to annotate a Java class field, a setter method, or a
1690 constructor parameter that is used to inject a service that resolves the reference. The interface of
1691 the service injected is defined by the type of the Java class field or the type of the input parameter
1692 of the setter method or constructor.

1693 The @Reference annotation MUST NOT be used on a class field that is declared as final.

1694 References can also be injected via setter methods even when the @Reference annotation is not
1695 present. However, the @Reference annotation must be used in order to inject a reference onto a
1696 non-public field. In the case where there is no @Reference annotation, the name of the reference
1697 is the same as the name of the field or setter.

Deleted: may

1698 Where there is both a setter method and a field for a reference, the setter method is used.

1699 The @Reference annotation has the following attributes:

- 1700 • **name (optional)** – the name of the reference. For a field annotation, the default is the
1701 name of the field of the Java class. For a setter method annotation, the default is the
1702 JavaBeans property name corresponding to the setter method name. For a constructor
1703 parameter annotation, there is no default and the name attribute MUST be present.
- 1704 • **required (optional)** – whether injection of service or services is required. Defaults to true.
1705 For a constructor parameter annotation, this attribute MUST have the value true.

1706

1707 The following snippet shows a reference field definition sample.

1708

```
1709 @Reference(name="stockQuote", required=true)  
1710 protected StockQuoteService stockQuote;
```

1711

1712 The following snippet shows a reference setter sample

1713

```
1714 @Reference(name="stockQuote", required=true)  
1715 public void setStockQuote( StockQuoteService theSQService ) {  
1716     ...  
1717 }
```

1718

1719 The following fragment from a component implementation shows a sample of a service reference
1720 using the @Reference annotation. The name of the reference is "helloService" and its type is
1721 HelloService. The clientMethod() calls the "hello" operation of the service referenced by the
1722 helloService reference.

1723

```
1724 package services.hello;  
1725  
1726 private HelloService helloService;  
1727  
1728 @Reference(name="helloService", required=true)  
1729 public setHelloService(HelloService service) {  
1730     helloService = service;  
1731 }  
1732  
1733 public void clientMethod() {  
1734     String result = helloService.hello("Hello World!");  
1735     ...  
1736 }  
1737
```

1738 The presence of a @Reference annotation is reflected in the componentType information that the
1739 runtime generates through reflection on the implementation class. The following snippet shows
1740 the component type for the above component implementation fragment.

1741

```
1742 <?xml version="1.0" encoding="ASCII"?>  
1743 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
1744     <!-- Any services offered by the component would be listed here -->
```

1745

```

1746     <reference name="helloService" multiplicity="1..1">
1747         <interface.java interface="services.hello.HelloService"/>
1748     </reference>
1749
1750 </componentType>
1751

```

1752 If the reference is not an array or collection, then the implied component type has a reference
1753 with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference **required**
1754 attribute – 1..1 applies if required=true.

1755

1756 If the reference is defined as an array or as any type that extends or implements **java.util.Collection**,
1757 then the implied component type has a reference with a **multiplicity** of either **1..n** or **0..n**, depending
1758 on whether the **required** attribute of the @Reference annotation is set to true or false – 1..n applies if
1759 required=true.

1760

1761 The following fragment from a component implementation shows a sample of a service reference
1762 definition using the @Reference annotation on a java.util.List. The name of the reference is
1763 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
1764 services referenced by the helloServices reference. In this case, at least one HelloService should
1765 be present, so **required** is true.

```

1766     @Reference(name="helloServices", required=true)
1767     protected List<HelloService> helloServices;
1768
1769     public void clientMethod() {
1770
1771         ...
1772         for (int index = 0; index < helloServices.size(); index++) {
1773             HelloService helloService =
1774                 (HelloService)helloServices.get(index);
1775             String result = helloService.hello("Hello World!");
1776         }
1777         ...
1778     }
1779
1780

```

1781 The following snippet shows the XML representation of the component type reflected from for the
1782 former component implementation fragment. There is no need to author this component type in
1783 this case since it can be reflected from the Java class.

1784

```

1785 <?xml version="1.0" encoding="ASCII"?>
1786 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1787
1788     <!-- Any services offered by the component would be listed here -->
1789     <reference name="helloServices" multiplicity="1..n">
1790         <interface.java interface="services.hello.HelloService"/>
1791     </reference>
1792
1793 </componentType>
1794

```

1795 At runtime, the representation of an unwired reference depends on the reference's multiplicity. An
1796 unwired reference with a multiplicity of 0..1 must be null. An unwired reference with a multiplicity
1797 of 0..N must be an empty array or collection.

1798

9.17.1 Reinjection

Formatted: Bullets and Numbering

1799 References MAY be reinjected after the initial creation of a component if the reference target
1800 changes due to a change in wiring that has occurred since the component was initialized. In order
1801 for reinjection to occur, the following MUST be true:

- 1802 1. The component MUST NOT be STATELESS scoped.
- 1803 2. The reference MUST use either field-based injection or setter injection. References that are
1804 injected through constructor injection MUST NOT be changed. Setter injection allows for
1805 code in the setter method to perform processing in reaction to a change.

1806 If a reference target changes and the reference is not reinjected, the reference MUST continue to
1807 work as if the reference target was not changed.

1808 If an operation is called on a reference where the target of that reference has been undeployed,
1809 the SCA runtime SHOULD throw InvalidServiceException. If an operation is called on a reference
1810 where the target of the reference has become unavailable for some reason, the SCA runtime
1811 SHOULD throw ServiceUnavailableException. If the target of the reference is changed, the
1812 reference MAY continue to work, depending on the runtime and the type of change that was made.
1813 If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.

1814 A ServiceReference that has been obtained from a reference by ComponentContext.cast()
1815 corresponds to the reference that is passed as a parameter to cast(). If the reference is
1816 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue
1817 to work as if the reference target was not changed. If the target of a ServiceReference has been
1818 undeployed, the SCA runtime SHOULD throw InvalidServiceException when an operation is
1819 invoked on the ServiceReference. If the target of a ServiceReference has become unavailable, the
1820 SCA runtime SHOULD throw ServiceUnavailableException when an operation is invoked on the
1821 ServiceReference. If the target of a ServiceReference is changed, the reference MAY continue to
1822 work, depending on the runtime and the type of change that was made. If it doesn't work, the
1823 exception thrown will depend on the runtime and the cause of the failure.

1824 A reference or ServiceReference accessed through the component context by calling getService()
1825 or getServiceReference() MUST correspond to the current configuration of the domain. This
1826 applies whether or not reinjection has taken place. If the target has been undeployed or has
1827 become unavailable, the result SHOULD be a reference to the undeployed or unavailable service,
1828 and attempts to call business methods SHOULD throw an exception as described above. If the
1829 target has changed, the result SHOULD be a reference to the changed service.

1830 The rules for reference reinjection also apply to references with a multiplicity of 0..N or 1..N. This
1831 means that in the cases listed above where reference reinjection is not allowed, the array or
1832 Collection for the reference MUST NOT change its contents. In cases where the contents of a
1833 reference collection MAY change, then for references that use setter injection, the setter method
1834 MUST be called for any change to the contents. The reinjected array or Collection MUST NOT be
1835 the same array or Collection object previously injected to the component.

1836

	Effect on		
Change event	Reference	Existing ServiceReference Object	Subsequent invocations of ComponentContext.getServiceReference() or getService()
Change to the target of the reference	MAY be reinjected (if other conditions* apply). If not reinjected, then it MUST continue to work as if the reference target was not changed.	MUST continue to work as if the reference target was not changed.	Result corresponds to the current configuration of the domain.
Target service	Business methods SHOULD throw	Business methods SHOULD throw	Result SHOULD be a reference to the undeployed

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

1837

1838

9.18 @Remotable

Formatted: Bullets and Numbering

1839

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

1840

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

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

1841

1842

1843

1844

1845

1846

1847

1848

1849

1850

1851

1852

1853

1854

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

1855

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

1856

The @Remotable annotation has no attributes.

1857

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

1860

```
package services.hello;

import org.oasisopen.sca.annotation.*;

@Remotable
public interface HelloService {

    String hello(String message);
}
```

1861

1862

1863

1864

1865

1866

1867

1868 }
1869

1870 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
1871 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

1872 Complex data types exchanged via remotable service interfaces MUST be compatible with the
1873 marshalling technology used by the service binding. For example, if the service is going to be
1874 exposed using the standard Web Service binding, then the parameters MAY be JAXB [JAX-B] types
1875 or Service Data Objects (SDOs) [SDO].

1876 Independent of whether the remotable service is called from outside of the composite that
1877 contains it or from another component in the same composite, the data exchange semantics are
1878 **by-value**.

1879 Implementations of remotable services can modify input data during or after an invocation and
1880 can modify return data after the invocation. If a remotable service is called locally or remotely, the
1881 SCA container is responsible for making sure that no modification of input data or post-invocation
1882 modifications to return data are seen by the caller.

Deleted: may

Deleted: may

1883 The following snippet shows a remotable Java service interface.

```
1884  
1885 package services.hello;  
1886  
1887 import org.oasisopen.sca.annotation.*;  
1888  
1889 @Remotable  
1890 public interface HelloService {  
1891     String hello(String message);  
1892 }  
1893  
1894 package services.hello;  
1895  
1896 import org.oasisopen.sca.annotation.*;  
1897  
1898 @Service(HelloService.class)  
1899 public class HelloServiceImpl implements HelloService {  
1900     public String hello(String message) {  
1901         ...  
1902     }  
1903 }  
1904 }  
1905
```

Formatted: Bullets and
Numbering

1906 **9.19 @Requires**

1907 The following Java code defines the **@Requires** annotation:

```
1908 package org.oasisopen.sca.annotation;  
1909  
1910 import static java.lang.annotation.ElementType.FIELD;  
1911 import static java.lang.annotation.ElementType.METHOD;  
1912 import static java.lang.annotation.ElementType.PARAMETER;  
1913 import static java.lang.annotation.ElementType.TYPE;  
1914 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1915  
1916 import java.lang.annotation.Inherited;  
1917 import java.lang.annotation.Retention;  
1918 import java.lang.annotation.Target;  
1919
```

```

1920 @Inherited
1921 @Retention(RUNTIME)
1922 @Target({TYPE, METHOD, FIELD, PARAMETER})
1923 public @interface Requires {
1924     /**
1925     * Returns the attached intents.
1926     *
1927     * @return the attached intents
1928     */
1929     String[] value() default "";
1930 }
1931
1932

```

The **@Requires** annotation supports general purpose intents specified as strings. User can also define specific intents using **@Intent** annotation.

See the section "General Intent Annotations" for details and samples.

Formatted: Bullets and Numbering

9.20 @Scope

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

```

1938 package org.oasisopen.sca.annotation;
1939
1940 import static java.lang.annotation.ElementType.TYPE;
1941 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1942 import java.lang.annotation.Retention;
1943 import java.lang.annotation.Target;
1944
1945 @Target(TYPE)
1946 @Retention(RUNTIME)
1947 public @interface Scope {
1948     String value() default "STATELESS";
1949 }
1950

```

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

Deleted: may

The **@Scope** annotation has the following attribute:

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

Deleted: may

Deleted: may

The default value is STATELESS.

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

```

1964 package services.hello;
1965
1966 import org.oasisopen.sca.annotation.*;
1967
1968 @Service(HelloService.class)
1969 @Scope("COMPOSITE")
1970 public class HelloServiceImpl implements HelloService {
1971
1972     public String hello(String message) {

```

1973
1974
1975
1976

```
...  
}  
}
```

1977

9.21 @Service

Formatted: Bullets and
Numbering

1978

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

1979

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

1980

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

1981

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

1982

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

1983

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

1984

```
@Target(TYPE)
```

1985

```
@Retention(RUNTIME)
```

1986

```
public @interface Service {
```

1987

```
    Class<?>[] interfaces() default {};
```

1988

```
    Class<?> value() default Void.class;
```

1989

```
}
```

1990

1991

1992

1993

1994

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

1995

1996

1997

1998

1999

2000

The @Service annotation has the following attributes:

2001

- **interfaces** – The value is an array of interface or class objects that should be exposed as services by this component.

2002

2003

- **value** – A shortcut for the case when the class provides only a single service interface.

2004

Only one of these attributes should be specified.

2005

2006

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

2007

2008

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

2009

2010

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

2011

2012

2013

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

2014

2015

```
package services.hello;
```

2016

```
import org.oasisopen.sca.annotation.Service;
```

2017

2018

```
@Service(HelloService.class)
```

2019

```
public class HelloServiceImpl implements HelloService {
```

2020

2021

```
    public void hello(String name) {
```

2022

```
2023         System.out.println("Hello " + name);
2024     }
2025 }
2026
```

Formatted: Bullets and Numbering

2027

10 WSDL to Java and Java to WSDL

2028
2029
2030

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

2031
2032
2033
2034
2035

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

2036
2037
2038
2039
2040

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

2041

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

2042

- No support for holders

2043

2044
2045

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

Formatted: Bullets and Numbering

2046

10.1 JAX-WS Client Asynchronous API for a Synchronous Service

2047

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

2052
2053
2054
2055
2056

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

2057

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

2058

a. a method name that is M's method name with the characters "Async" appended, and

2059

b. the same parameter signature as M, and

2060

c. a return type of Response<R> where R is the return type of M

2061

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

2062

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

2063

a. a method name that is M's method name with the characters "Async" appended, and

2064

b. a parameter signature that is M's parameter signature with an additional final parameter of type AsyncHandler<R> where R is the return type of M, and

2065

c. a return type of Future<?>

2066

2067

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

2068

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

Deleted: may

2069

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

2070

```
2071 <part name="ticker" type="xsd:string"/>
2072 </message>
2073
2074 <message name="getPriceResponse">
2075 <part name="price" type="xsd:float"/>
2076 </message>
2077
2078 <portType name="StockQuote">
2079 <operation name="getPrice">
2080 <input message="tns:getPrice"/>
2081 <output message="tns:getPriceResponse"/>
2082 </operation>
2083 </portType>
```

2084

2085 The JAX-WS asynchronous mapping will produce the following Java interface:

```
2086 // asynchronous mapping
2087 @WebService
2088 public interface StockQuote {
2089     float getPrice(String ticker);
2090     Response<Float> getPriceAsync(String ticker);
2091     Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
2092 }
```

2093

2094 For SCA interface definition purposes, this is treated as equivalent to the following:

```
2095 // synchronous mapping
2096 @WebService
2097 public interface StockQuote {
2098     float getPrice(String ticker);
2099 }
```

2100

2101 SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above
2102 example, if the client implementation uses the asynchronous form of the interface, the two
2103 additional getPriceAsync() methods can be used for polling and callbacks as defined by the JAX-
2104 WS specification.

2105

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

```
2106 <?xml version="1.0" encoding="UTF-8"?>
2107 <!-- (c) Copyright SCA Collaboration 2006 -->
2108 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2109         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2110         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2111         elementFormDefault="qualified">
2112
2113     <include schemaLocation="sca-core.xsd"/>
2114
2115     <element name="interface.java" type="sca:JavaInterface"
2116             substitutionGroup="sca:interface"/>
2117     <complexType name="JavaInterface">
2118         <complexContent>
2119             <extension base="sca:Interface">
2120                 <sequence>
2121                     <any namespace="##other" processContents="lax"
2122                         minOccurs="0" maxOccurs="unbounded"/>
2123                 </sequence>
2124                 <attribute name="interface" type="NCName" use="required"/>
2125                 <attribute name="callbackInterface" type="NCName"
2126                         use="optional"/>
2127                 <anyAttribute namespace="##any" processContents="lax"/>
2128             </extension>
2129         </complexContent>
2130     </complexType>
2131 </schema>
```

2132 **B. Conformance Items**

2133 This section contains a list of conformance items for the SCA Java Common Annotations and APIs
2134 specification.

2135

Conformance ID	Description
[JCA30001]	@interface MUST be the fully qualified name of the Java interface class
[JCA30002]	@callbackInterface MUST be the fully qualified name of a Java interface used for callbacks
[JCA30003]	However, if the Java interface class identified by the @interface attribute does contain a Java @Callback annotation, then the Java interface class identified by the @callbackInterface attribute MUST be the same interface class.

2136

2137

C. Acknowledgements

2138 The following individuals have participated in the creation of this specification and are gratefully
2139 acknowledged:

2140 **Participants:**

2141 [Participant Name, Affiliation | Individual Member]

2142 [Participant Name, Affiliation | Individual Member]

2143

D. Non-Normative Text

2145

E. Revision History

2146 [optional; should not be included in OASIS Standards]

2147

Revision	Date	Editor	Changes Made
1	2007-09-26	Anish Karmarkar	Applied the OASIS template + related changes to the Submission
2	2008-02-28	Anish Karmarkar	Applied resolution of issues: 4, 11, and 26
3	2008-04-17	Mike Edwards	Ed changes
4	2008-05-27	Anish Karmarkar David Booz Mark Combella	Added InvalidServiceException in Section 7 Various editorial updates
WD04	2008-08-15	Anish Karmarkar	* Applied resolution of issue 9 (it was applied before, not sure by whom, but it was applied incorrectly) * Applied resolution of issue 12, 22, 23, 29, 31, 35, 36, 37, 44, 45 * Note that issue 33 was applied, but not noted, in a previous version * Replaced the osoa.org NS with the oasis-open.org NS
WD05	2008-10-03	Anish Karmarkar	* Fixed the resolution of issue 37 but re-adding the sentence: "However, the @... annotation must be used in order to inject a property onto a non-public field. -- in the @Property and @Reference section * resolution of issue 9 was applied incorrectly. Fixed that -- removed the requirement for throwing an exception on ComponentContext.getServiceReferences() when multiplicity of references > 1 * minor ed changes
cd01-rev1	2008-12-11	Anish Karmarkar	* Fixed reference style to [RFC2119] instead of [1]. * Applied resolutions of issues 20, 21, 41, 42, 43, 47, 48, 49.
cd01-rev2	2008-12-12	Anish Karmarkar	* Applied resolutions of issues 61, 71, 72, 73, 79, 81, 82, 84, 112
cd01-rev3	2008-12-16	David Booz	* Applied resolution of issues 56, 75, 111
cd01-rev4	2009-01-18	Anish Karmarkar	* Applied resolutions of issues 28, 52, 94, 96, 99, 101
cd02	2009-01-26	Mike Edwards	Minor editorial cleanup. All changes accepted.

			All comments removed.
cd02-rev1	2009-02-03	Mike Edwards	Issues 25+95 Issue 120

2148