# OASIS

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

## Committee Draft 02, Revision 01+AnnotationsMerge+Issue27

## 03 February 2009

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

**Previous Version:**

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

**Latest Approved Version:**

**Technical Committee:**
> OASIS Service Component Architecture / J (SCA-J) TC

**Chair(s):**
> Simon Nash,          IBM
> MIchael Rowley,      BEA Systems
> Mark Combellack,     Avaya

**Editor(s):**
> Ron Barack,          SAP
> David Booz,          IBM
> Mark Combellack,     Avaya
> Mike Edwards,        IBM
> Anish Karmarkar,     Oracle
> Ashok Malhotra,      Oracle
> Peter Peshev,        SAP

**Related work:**
> This specification replaces or supersedes:
> - Service Component Architecture Java Annotations and APIs Specification Version 1.00, March 21 2007
>
> This specification is related to:
> - Service Component Architecture Assembly Model Specification Version 1.1

- Service Component Architecture Policy Framework Specification Version 1.1

**Declared XML Namespace(s):**

**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 maycan be used by Java-based SCA specifications.

Specifically, this specification covers:

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

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

**Status:**

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

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

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

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

# Notices

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

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

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

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

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

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

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

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

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

# Table of Contents

# 1 Introduction

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 maycan be used by Java-based SCA specifications.

Specifically, this specification covers:

1. Implementation metadata for specifying component services, references, and properties

2. A client and component API

3. Metadata for asynchronous services

4. Metadata for callbacks

5. Definitions of standard component implementation scopes

6. Java to WSDL and WSDL to Java mappings

7. Security policy annotations

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

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

This document defines implementation metadata using the annotation capability from Java$^{TM}$ 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, *Key words for use in RFCs to Indicate Requirement Levels*, 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 |

## 2 Implementation Metadata

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

### 2.1 Service Metadata

#### 2.1.1 @Service

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

- As a Java interface
- As a Java class
- As a Java interface generated from a Web Services Description Language [WSDL] (WSDL) portType (Java interfaces generated from a WSDL portType are always **remotable**)

#### 2.1.2 Java Semantics of a Remotable Service

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

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

```
package services.hello;
@Remotable
public interface HelloService {
    String hello(String message);
}
```

#### 2.1.3 Java Semantics of a Local Service

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

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

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

```
package services.hello;
public interface HelloService {
    String hello(String message);
}
```

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

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

## 2.1.4 @Reference

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

## 2.1.5 @Property

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

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

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

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

This document defines two scopes:

- STATELESS
- COMPOSITE

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

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

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

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

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

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

```
@Init
public void start() {
        ...
}

@Destroy
public void stop() {
        ...
}
```

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

## 2.2.1 Stateless scope

For stateless scope components, there is no implied correlation between implementation instances 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.

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

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

## 3  Interface

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

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

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

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

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

The interface.java element has the following attributes:

- **interface (1..1)** – the Java interface class to use for the service interface. @interface MUST be the fully qualified name of the Java interface class [JCA30001]

- **callbackInterface (0..1)** – the Java interface class to use for the callback interface. @callbackInterface MUST be the fully qualified name of a Java interface used for callbacks [JCA30002]

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

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

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

Note that the Java interface class identified by the @interface attribute can contain a Java @Callback annotation which identifies a callback interface. If this is the case, then it is not necessary to provide the @callbackInterface attribute. 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. [JCA30003]

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

## 3.2 @Remotable

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

## 3.3 @Callback

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

# 4  Client API

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

## 4.1 Accessing Services from an SCA Component

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

### 4.1.1 Using the Component Context API

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

## 4.2 Accessing Services from non-SCA component implementations

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

### 4.2.1 ComponentContext

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

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

229

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

## 5  Error Handling

Clients calling service methods maycan experience business exceptions and SCA runtime exceptions.

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

SCA runtime exceptions are raised by the SCA runtime and signal problems in management of component execution or problems interacting with remote services. The SCA runtime exceptions are defined in the Java API section.

# 6 Asynchronous Programming

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

- support for non-blocking method calls
- callbacks

Each of these topics is discussed in the following sections.

## 6.1 @OneWay

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

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

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

## 6.2 Callbacks

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

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

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

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

### 6.2.1 Using Callbacks

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

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

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

```
289  package somepackage;
290  import org.osoa.sca.annotation.Callback;
291  import org.osoa.sca.annotation.Remotable;
292  @Remotable
293  @Callback(QuotationCallback.class)
294  public interface Quotation {h
295      double requestQuotation(String productCode, int quantity);
296  }
297
298  @Remotable
299  public interface QuotationCallback {
300      String getState();
301      String getZipCode();
302      String getCreditRating();
303  }
304
```

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

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

```
314
315  @Callback
316  protected QuotationCallback callback;
317
318  public double requestQuotation(String productCode, int quantity) {
319      double price = getPrice(productQuote, quantity);
320      double discount = 0;
321      if (quantity > 1000 && callback.getState().equals("FL")) {
322          discount = 0.05;
323      }
324      if (quantity > 10000 && callback.getCreditRating().charAt(0) == 'A') {
325          discount += 0.05;
326      }
327      return price * (1-discount);
328  }
329
```

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

```
333
334  public class ClientImpl implements ClientService, QuotationCallback {
335
336      private QuotationService myService;
337
338      @Reference
339      public void setMyService(QuotationService service) {
340          myService = service;
341      }
```

```
342
343         public void aClientMethod() {
344             ...
345             double quote = myService.requestQuotation("AB123", 2000);
346             ...
347         }
348
349         public String getState() {
350             return "TX";
351         }
352         public String getZipCode() {
353             return "78746";
354         }
355         public String getCreditRating() {
356             return "AA";
357         }
358     }
359
```

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

## 364  6.2.2 Callback Instance Management

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

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

## 376  6.2.3 Implementing Multiple Bidirectional Interfaces

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

```
383
384     @Callback
385     protected MyService1Callback callback1;
386
387     @Callback
388     protected MyService2Callback callback2;
389
```

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

## 6.2.4 Accessing Callbacks

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

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

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

```
@Callback
protected ServiceReference<MyCallback> callback;

public void someMethod() {

    MyCallback myCallback = callback.getCallback();    …

    myCallback.receiveResult(theResult);
}
```

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

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

```
public void someMethod() {

    MyCallback myCallback =
         ComponentContext.getRequestContext().getCallback();

        …

    myCallback.receiveResult(theResult);
}
```

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

```
@Context
protected RequestContext requestContext;

void receiveResult(Object theResult) {

    Object refParams =
         requestContext.getServiceReference().getCallbackID();
        …
}
```

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

# 7 Policy Annotations for Java

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

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

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

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

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

## 7.1 General Intent Annotations

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

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

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

"{" + *Namespace URI* + "}" + *intentname*

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

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

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

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

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

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

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

495

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

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

```
498     package com.foo;
499
500     import static org.oasisopen.sca.annotation.Confidentiality.*;
501     import static org.oasisopen.sca.annotation.Reference;
502     import static org.oasisopen.sca.annotation.Requires;
503
504
505     public class Foo {
506         @Requires(CONFIDENTIALITY)
507         @Reference
508         public void setBar(Bar bar) {
509             …
510         }
511     }
512
```

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

```
516     package com.foo;
517
518     import static org.oasisopen.sca.Constants.*;
519     import static org.oasisopen.sca.annotation.Reference;
520     import static org.oasisopen.sca.annotation.Requires;
521
522     public class Foo {
523         @Requires(SCA_PREFIX+"confidentiality")
524         @Reference
525         public void setBar(Bar bar) {
526             …
527         }
528     }
529
```

530 The formal syntax for the @Requires annotation follows:

531     @Requires( "qualifiedIntent" (, "qualifiedIntent")* )

532 where

533     qualifiedIntent ::= QName(.qualifier)*

534

535 See section @Requires for the formal definition of the @Requires annotation.

## 7.2 Specific Intent Annotations

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.

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.

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:

```
@Integrity
```

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

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

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").

The general form of specific intent annotations is:

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

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

```
Intent        ::= NCName
qualifiers    ::= "qualifier" (, "qualifier")*
qualifier     ::= NCName(.qualifier)?
```

## 7.2.1 How to Create Specific Intent Annotations

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

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.

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

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

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

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.

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:

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

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

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

584   Examples of the use of the @Intent and the @Qualifier annotations in the definition of specific
585   intent annotations are shown in the section dealing with Security Interaction Policy.

## 7.3 Application of Intent Annotations

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

588   • Java class

589   • Java interface

590   • Method

591   • Field

592   • Constructor parameter

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

595       @Authentication
596       @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})

597   In this case, the effective intents are "authentication", "confidentiality.message" and
598   "integrity.message".

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

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

## 7.3.1 Inheritance And Annotation

606   The inheritance rules for annotations are consistent with the common annotation specification, JSR
607   250.

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

```
610       package services.hello;
611       import org.oasisopen.sca.annotation.Remotable;
612       import org.oasisopen.sca.annotation.Integrity;
613       import org.oasisopen.sca.annotation.Authentication;
614
615       @Integrity("transport")
616       @Authentication
617       public class HelloService {
618             @Integrity
619             @Authentication("message")
620             public String hello(String message) {...}
621
622             @Integrity
623             @Authentication("transport")
624             public String helloThere() {...}
625       }
626
627       package services.hello;
628       import org.oasisopen.sca.annotation.Remotable;
629       import org.oasisopen.sca.annotation.Confidentiality;
630       import org.oasisopen.sca.annotation.Authentication;
```

```
631
632        @Confidentiality("message")
633        public class HelloChildService extends HelloService {
634                @Confidentiality("transport")
635                public String hello(String message) {...}
636                @Authentication
637                String helloWorld() {...}
638        }
```
Example 2a.  Usage example of annotated policy and inheritance.

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

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

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

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

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

```
654        <?xml version="1.0" encoding="ASCII"?>
655        <composite  xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
656                    name="HelloServiceComposite" >
657            <service name="HelloService" requires="integrity/transport
658                authentication">
659                …
660            </service>
661            <service name="HelloChildService" requires="integrity/transport
662                authentication confidentiality/message">
663            …
664            </service>
665            ...
666
667            <component name="HelloServiceComponent">*
668                <implementation.java class="services.hello.HelloService"/>
669                        <operation name="hello" requires="integrity
670                            authentication/message"/>
671                        <operation name="helloThere"
672                                requires="integrity
673                                authentication/transport"/>
674            </component>
675            <component name="HelloChildServiceComponent">*
676                <implementation.java
677                    class="services.hello.HelloChildService" />
678                <operation name="hello"
679                    requires="confidentiality/transport"/>
680                <operation name="helloThere" requires=" integrity/transport
681                    authentication"/>
682                <operation name=helloWorld" requires="authentication"/>
683            </component>
684
```

```
685            . . .
686
687        </composite>
688
689    Example 2b.  Declaratives intents equivalent to annotated intents in Example 2a.
690
```

## 7.4 Relationship of Declarative And Annotated Intents

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

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

## 7.5 Policy Set Annotations

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

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

```
707        @PolicySets( "<policy set QName>" |
708                    { "<policy set QName>" [, "<policy set QName>"] })
709
```

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

An example of the @PolicySets annotation:

```
712
713        @Reference(name="helloService", required=true)
714        @PolicySets({ MY_NS + "WS_Encryption_Policy",
715                    MY_NS + "WS_Authentication_Policy" })
716        public setHelloService(HelloService service) {
717            . . .
718        }
719
```

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

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

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

- Java class
- Java interface
- Method
- Field
- Constructor parameter

## 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, @Confidentality 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!");
```

```
780                      …
781                  }
782              }
783

784          Example 1.  Usage of annotated intents on a reference.
```

## 7.6.2 Security Implementation Policy

SCA ~~defines~~java implementation honors the set ~~a number~~ of security policy annotations that apply as policies to implementations themselves. These annotations mostly have to do with authorization and security identity. The following authorization and security identity annotations (as defined in JSR 250) are supported:

- RunAs

  Takes as a parameter a string which is the name of a Security role.
  e.g., @RunAs("Manager")

  Code marked with this annotation will execute with the Security permissions of the identified role. The @RunAs annotations can be mapped to <runAs> element that is defined in the policy specification. Any code so annotated will run with the permissions of that role. How runAs role names are mapped to security principals is implementation dependent. e.g., the above @RunAs annotation can be mapped as if the following policySet is defined and attached to the level where the annotation applies:
  ```
  <policySet name="runas_manager">
      <securityIdentity>
          <runAs role="Manager"/>
      </securityIdentity>
  </policySet>
  ```

- RolesAllowed

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

  The @RolesAllowed annotation can be mapped to <allow> element that is defined in the policy specification. It indicates that access is granted only to principals whose role corresponds to one of the role names listed in the @roles attribute. How role names are mapped to security principals is SCA Runtime implementation dependent. e.g., the above @RolesAllowed annotation can be mapped as if the following policySet is defined and attached to the level where the annotation applies:
  ```
  <policySet name="allow_manager_employee">
      <authorization>
          <allow roles="Manager Employee"/>
      </authorization>
  </policySet>
  ```

- PermitAll

  No parameters.  When present, grants access to all roles.
  The @PermitAll annotation can be mapped to <permitAll> element that is defined in the policy specification. It indicates to grant access to all principals respectively. e.g., a @PermitAll annotation can be mapped as if the following policySet is defined and attached to the level where the annotation applies:
  ```
  <policySet name="permitAll">
      <authorization>
          <permitAll/>
  ```

835         `</authorization>`
836     `</policySet>`
837

- DenyAll

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

841 The @DenyAll annotation can be mapped to <denyAll> element that is defined in the
842 policy specification. It indicates to deny access to all principals respectively. e.g., a
843 @DenyAll annotation can be mapped as if the following policySet is defined and attached
844 to the level where the annotation applies:
845 `<policySet name="denyAll">`
846     `<authorization>`
847       `<denyAll/>`
848     `</authorization>`
849 `</policySet>`
850

- DeclareRoles

852 Takes as a parameter a string or an array of strings which identify one or more role names
853 that form the set of roles used by the implementation.
854 e.g., @DeclareRoles({"Manager", "Employee", "Customer"} )

856 There is no mapping to elements defined in policy specifications.

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

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

## 7.6.2.1 Annotated Implementation Policy Example

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

```
package services.account;
@Remotable
public interface AccountService {
        AccountReport getAccountReport(String customerID);
        float getCurrencyFromUSD(float value);
}
```

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

```
package services.account;
import java.util.List;
import commonj.sdo.DataFactory;
import org.oasisopen.sca.annotation.Property;
import org.oasisopen.sca.annotation.Reference;
import
org.oasisopen.sca.annotation.javax.annotation.security.RolesAllowed;
import javax.annotation.security.org.oasisopen.sca.annotation.RunAs;
```

```
884   import javax.annotation.security.org.oasisopen.sca.annotation.PermitAll;
885   import services.accountdata.AccountDataService;
886   import services.accountdata.CheckingAccount;
887   import services.accountdata.SavingsAccount;
888   import services.accountdata.StockAccount;
889   import services.stockquote.StockQuoteService;
890   @RolesAllowed("customers")
891   @RunAs("accountants" )
892   public class AccountServiceImpl implements AccountService {
893
894       @Property
895       protected String currency = "USD";
896
897       @Reference
898       protected AccountDataService accountDataService;
899       @Reference
900       protected StockQuoteService stockQuoteService;
901
902       @RolesAllowed({"customers", "accountants"})
903       public AccountReport getAccountReport(String customerID) {
904
905        DataFactory dataFactory = DataFactory.INSTANCE;
906        AccountReport accountReport =
907             (AccountReport)dataFactory.create(AccountReport.class);
908        List accountSummaries = accountReport.getAccountSummaries();
909
910        CheckingAccount checkingAccount =
911             accountDataService.getCheckingAccount(customerID);
912        AccountSummary checkingAccountSummary =
913             (AccountSummary)dataFactory.create(AccountSummary.class);
914
915   checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber()
916   );
917        checkingAccountSummary.setAccountType("checking");
918        checkingAccountSummary.setBalance(fromUSDollarToCurrency
919             (checkingAccount.getBalance()));
920        accountSummaries.add(checkingAccountSummary);
921
922        SavingsAccount savingsAccount =
923             accountDataService.getSavingsAccount(customerID);
924        AccountSummary savingsAccountSummary =
925             (AccountSummary)dataFactory.create(AccountSummary.class);
926
927   savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
928        savingsAccountSummary.setAccountType("savings");
929        savingsAccountSummary.setBalance(fromUSDollarToCurrency
930             (savingsAccount.getBalance()));
931        accountSummaries.add(savingsAccountSummary);
932
933        StockAccount stockAccount =
934   accountDataService.getStockAccount(customerID);
935        AccountSummary stockAccountSummary =
936             (AccountSummary)dataFactory.create(AccountSummary.class);
937        stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
938        stockAccountSummary.setAccountType("stock");
939        float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*
940                     stockAccount.getQuantity();
941        stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
```

```
942          accountSummaries.add(stockAccountSummary);
943
944          return accountReport;
945      }
946
947      @PermitAll
948      public float getCurrencyFromUSD(float value){
949
950          Return fromUSDollarToCurrency(value);
951
952      }
953
954      public float fromUSDollarToCurrency(float value) {
955
956          if (currency.equals("USD")) return value; else
957          if (currency.equals("EURO")) return value * 0.8f; else
958          return 0.0f;
959      }
960  }
```

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

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

963 • @RolesAllowed("customers")  - indicating that customers have access to the
964   implementation as a whole

965 • @RunAs("accountants" ) – indicating that the code in the implementation runs with the
966   permissions of accountants

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

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

## 78 Java API

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

### 7.18.1 Component Context

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

```
package org.oasisopen.sca;

public interface ComponentContext {

    String getURI();

    <B> B getService(Class<B> businessInterface, String referenceName);

    <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
                                                String referenceName);
    <B> Collection<B> getServices(Class<B> businessInterface,
        String referenceName);

    <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
        businessInterface, String referenceName);

    <B> ServiceReference<B> createSelfReference(Class<B>
        businessInterface);

    <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
                                                String serviceName);

    <B> B getProperty(Class<B> type, String propertyName);

    <B, R extends ServiceReference<B>> R cast(B target)
                    throws  IllegalArgumentException;

    RequestContext getRequestContext();


}
```

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

- **getServices(Class<B> businessInterface, String referenceName)** – Returns a list of typed service proxies for a business interface type and a reference name.

- **getServiceReferences(Class<B> businessInterface, String referenceName)** –Returns a list typed service references for a business interface type and a reference name.

- **createSelfReference(Class<B> businessInterface)** – Returns a ServiceReference that can be used to invoke this component over the designated service.

- **createSelfReference(Class<B> businessInterface, String serviceName)** – Returns a ServiceReference that can be used to invoke this component over the designated service. Service name explicitly declares the service name to invoke

- **getProperty** (**Class<B> type, String propertyName**) - Returns the value of an SCA property defined by this component.

- **getRequestContext()** - Returns the context for the current SCA service request, or null if there is no current request or if the context is unavailable. This method MUST return non-null when invoked during the execution of a Java business method for a service operation or callback operation, on the same thread that the SCA runtime provided, and MUST return null in all other cases.

- **cast**(**B target**) - Casts a type-safe reference to a ServiceReference

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

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

```
private ComponentContext componentContext;

@Context
public void setContext(ComponentContext context) {
    componentContext = context;
}

public void doSomething() {
    HelloWorld service =
    componentContext.getService(HelloWorld.class,"HelloWorldComponent");
    service.hello("hello");
}
```

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

## 7.28.2 Request Context

The following shows the **RequestContext** interface:

```
package org.oasisopen.sca;

import javax.security.auth.Subject;

public interface RequestContext {

    Subject getSecuritySubject();

    String getServiceName();
```

```
1069        <CB> ServiceReference<CB> getCallbackReference();
1070        <CB> CB getCallback();
1071        <B> ServiceReference<B> getServiceReference();
1072
1073    }
1074
```

1075    The RequestContext interface has the following methods:

1076    • **getSecuritySubject()** – Returns the JAAS Subject of the current request

1077    • **getServiceName()** – Returns the name of the service on the Java implementation the
1078        request came in on

1079    • **getCallbackReference()** – Returns a service reference to the callback as specified by the
1080        caller. This method returns null when called for a service request whose interface is not
1081        bidirectional or when called for a callback request.

1082    • **getCallback()** – Returns a proxy for the callback as specified by the caller. Similar to the
1083        getCallbackReference() method, this method returns null when called for a service request
1084        whose interface is not bidirectional or when called for a callback request.

1085    • **getServiceReference()** – When invoked during the execution of a service operation, this
1086        method MUST return a ServiceReference that represents the service that was invoked.
1087        When invoked during the execution of a callback operation, this method MUST return a
1088        CallableReference that represents the callback that was invoked.

## 7.38.3 ServiceReference

1090    ServiceReferences maycan be injected using the @Reference annotation on a field,  a setter
1091    method, or constructor parameter taking the type ServiceReference.  The detailed description of
1092    the usage of these methods is described in the section on Asynchronous Programming in this
1093    document.

1094    The following Java code defines the **ServiceReference** interface:

```
1095    package org.oasisopen.sca;
1096
1097    public interface ServiceReference<B> extends java.io.Serializable {
1098
1099        B getService();
1100        Class<B> getBusinessInterface();
1101    }
1102
```

1103    The ServiceReference interface has the following methods:

1104

1105    • **getService()**  - Returns a type-safe reference to the target of this reference.  The instance
1106        returned is guaranteed to implement the business interface for this reference.  The value
1107        returned is a proxy to the target that implements the business interface associated with this
1108        reference.

1109    • **getBusinessInterface()** – Returns the Java class for the business interface associated with
1110        this reference.

## 7.48.4 ServiceRuntimeException

1112    The following snippet shows the **ServiceRuntimeException**.

1113

```
1114    package org.oasisopen.sca;
1115
```

```
1116    public class ServiceRuntimeException extends RuntimeException {
1117        …
1118    }
1119
1120    This exception signals problems in the management of SCA component execution.
```

### 7.58.5 ServiceUnavailableException

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

```
1123
1124    package org.oasisopen.sca;
1125
1126    public class ServiceUnavailableException extends ServiceRuntimeException {
1127        …
1128    }
1129
```

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

### 7.68.6 InvalidServiceException

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

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

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

## 8.7 Constants Interface

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

```
1149        package org.oasisopen.sca;
1150
1151        public interface Constants {
1152            String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";
1153            String SCA_PREFIX = "{"+SCA_NS+"}";
1154        }
1155
```

## 89 Java Annotations

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

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

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

### 8.19.1 @AllowsPassByReference

The following Java code defines the ***@AllowsPassByReference*** 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.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

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

}
```

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

@AllowsPassByReference has no attributes

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

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

## 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.

## 8.29.3 @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;
```

```
1254        }
1255
1256
1257     The @Callback annotation is used to annotate a service interface with a callback interface, which
1258     takes the Java Class object of the callback interface as a parameter.

1259     The @Callback annotation has the following attribute:

1260        •    value – the name of a Java class file containing the callback interface

1261

1262     The @Callback annotation maycan also be used to annotate a method or a field of an SCA
1263     implementation class, in order to have a callback object injected

1264

1265     The following snippet shows a @Callback annotation on an interface:

1266
```

```
1267     @Remotable
1268     @Callback(MyServiceCallback.class)
1269     public interface MyService {
1270
1271         void someAsyncMethod(String arg);
1272     }
1273
```

1274     An example use of the @Callback annotation to declare a callback interface follows:

1275

```
1276     package somepackage;
1277     import org.oasisopen.sca.annotation.Callback;
1278     import org.oasisopen.sca.annotation.Remotable;
1279     @Remotable
1280     @Callback(MyServiceCallback.class)
1281     public interface MyService {
1282
1283         void someMethod(String arg);
1284     }
1285
1286     @Remotable
1287     public interface MyServiceCallback {
1288
1289         void receiveResult(String result);
1290     }
1291
```

1292     In this example, the implied component type is:

1293

```
1294     <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >
1295
1296         <service name="MyService">
1297             <interface.java interface="somepackage.MyService"
1298                     callbackInterface="somepackage.MyServiceCallback"/>
1299         </service>
1300     </componentType>
```

## 8.39.4 @ComponentName

1302     The following Java code defines the *@ComponentName* annotation:

```
1303
1304    package org.oasisopen.sca.annotation;
1305
1306    import static java.lang.annotation.ElementType.METHOD;
1307    import static java.lang.annotation.ElementType.FIELD;
1308    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1309    import java.lang.annotation.Retention;
1310    import java.lang.annotation.Target;
1311
1312    @Target({METHOD, FIELD})
1313    @Retention(RUNTIME)
1314    public @interface ComponentName {
1315
1316    }
1317
```

1318 The @ComponentName annotation is used to denote a Java class field or setter method that is
1319 used to inject the component name.

1320 The following snippet shows a component name field definition sample.

```
1321
1322    @ComponentName
1323    private String componentName;
1324
```

1325 The following snippet shows a component name setter method sample.

```
1326
1327    @ComponentName
1328    public void setComponentName(String name) {
1329      //…
1330    }
```

## 9.5 @Confidentiality

1332 The following Java code defines the **@Confidentiality** annotation:

```
1333
1334    package org.oasisopen.sca.annotations;
1335
1336    import static java.lang.annotation.ElementType.FIELD;
1337    import static java.lang.annotation.ElementType.METHOD;
1338    import static java.lang.annotation.ElementType.PARAMETER;
1339    import static java.lang.annotation.ElementType.TYPE;
1340    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1341    import static org.oasisopen.sca.Constants.SCA_PREFIX;
1342
1343    import java.lang.annotation.Inherited;
1344    import java.lang.annotation.Retention;
1345    import java.lang.annotation.Target;
1346
1347    @Inherited
1348    @Target({TYPE, FIELD, METHOD, PARAMETER})
1349    @Retention(RUNTIME)
1350    @Intent(Confidentiality.CONFIDENTIALITY)
1351    public @interface Confidentiality {
1352        String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
1353        String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
1354        String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";
```

```
1355
1356        /**
1357         * List of confidentiality qualifiers (such as "message" or
1358     "transport").
1359         *
1360         * @return confidentiality qualifiers
1361         */
1362        @Qualifier
1363        String[] value() default "";
1364     }
```

1365 The **@Confidentiality** annotation is used to indicate that the invocation requires confidentiality.

1366 See the section on Application of Intent Annotations for samples and details.

### 1367 8.49.6 @Constructor

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

```
1369
1370     package org.oasisopen.sca.annotation;
1371
1372     import static java.lang.annotation.ElementType.CONSTRUCTOR;
1373     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1374     import java.lang.annotation.Retention;
1375     import java.lang.annotation.Target;
1376
1377     @Target(CONSTRUCTOR)
1378     @Retention(RUNTIME)
1379     public @interface Constructor { }
1380
```

1381 The @Constructor annotation is used to mark a particular constructor to use when instantiating a
1382 Java component implementation. If this constructor has parameters, each of these parameters
1383 MUST have either a @Property annotation or a @Reference annotation.

1384 The following snippet shows a sample for the @Constructor annotation.

```
1385
1386     public class HelloServiceImpl implements HelloService {
1387
1388        public HelloServiceImpl(){
1389         ...
1390         }
1391
1392        @Constructor
1393        public HelloServiceImpl(@Property(name="someProperty") String
1394     someProperty ){
1395         ...
1396         }
1397
1398         public String hello(String message) {
1399            ...
1400          }
1401     }
```

### 1402 8.59.7 @Context

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

1404

```
1405    package org.oasisopen.sca.annotation;
1406
1407    import static java.lang.annotation.ElementType.METHOD;
1408    import static java.lang.annotation.ElementType.FIELD;
1409    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1410    import java.lang.annotation.Retention;
1411    import java.lang.annotation.Target;
1412
1413    @Target({METHOD, FIELD})
1414    @Retention(RUNTIME)
1415    public @interface Context {
1416
1417    }
1418
```

1419   The @Context annotation is used to denote a Java class field or a setter method that is used to
1420   inject a composite context for the component. The type of context to be injected is defined by the
1421   type of the Java class field or type of the setter method input argument; the type is either
1422   **ComponentContext** or **RequestContext**.

1423   The @Context annotation has no attributes.

1424

1425   The following snippet shows a ComponentContext field definition sample.

1426

```
1427    @Context
1428    protected ComponentContext context;
1429
```

1430   The following snippet shows a RequestContext field definition sample.

1431

```
1432    @Context
1433    protected RequestContext context;
```

## 8.69.8 @Destroy

1435   The following Java code defines the **@Destroy** annotation:

1436

```
1437    package org.oasisopen.sca.annotation;
1438
1439    import static java.lang.annotation.ElementType.METHOD;
1440    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1441    import java.lang.annotation.Retention;
1442    import java.lang.annotation.Target;
1443
1444    @Target(METHOD)
1445    @Retention(RUNTIME)
1446    public @interface Destroy {
1447
1448    }
1449
```

1450   The @Destroy annotation is used to denote a single Java class method that will be called when the
1451   scope defined for the implementation class ends. The method MAY have any access modifier and
1452   MUST have a void return type and no arguments.

1453   If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1454   when the scope defined for the implementation class ends. If the implementation class has a

1455 method with an @Destroy annotation that does not match these criteria, the SCA runtime MUST
1456 NOT instantiate the implementation class.

1457 The following snippet shows a sample for a destroy method definition.

1458

```
1459   @Destroy
1460   public void myDestroyMethod() {
1461       …
1462   }
```

### 8.79.9 @EagerInit

1464 The following Java code defines the *@EagerInit* annotation:

1465

```
1466   package org.oasisopen.sca.annotation;
1467
1468   import static java.lang.annotation.ElementType.TYPE;
1469   import static java.lang.annotation.RetentionPolicy.RUNTIME;
1470   import java.lang.annotation.Retention;
1471   import java.lang.annotation.Target;
1472
1473   @Target(TYPE)
1474   @Retention(RUNTIME)
1475   public @interface EagerInit {
1476
1477   }
```

1478

1479 The *@EagerInit* annotation is used to annotate the Java class of a COMPOSITE scoped
1480 implementation for eager initialization. When marked for eager initialization, the composite scoped
1481 instance is created when its containing component is started.

### 8.89.10 @Init

1483 The following Java code defines the *@Init* annotation:

1484

```
1485   package org.oasisopen.sca.annotation;
1486
1487   import static java.lang.annotation.ElementType.METHOD;
1488   import static java.lang.annotation.RetentionPolicy.RUNTIME;
1489   import java.lang.annotation.Retention;
1490   import java.lang.annotation.Target;
1491
1492   @Target(METHOD)
1493   @Retention(RUNTIME)
1494   public @interface Init {
1495
1496
1497   }
```

1498

1499 The @Init annotation is used to denote a single Java class method that is called when the scope
1500 defined for the implementation class starts. The method MAY have any access modifier and MUST
1501 have a void return type and no arguments.

1502 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1503 after all property and reference injection is complete. If the implementation class has a method

1504 with an @Init annotation that does not match these criteria, the SCA runtime MUST NOT
1505 instantiate the implementation class.

1506 The following snippet shows an example of an init method definition.

1507

```
1508    @Init
1509    public void myInitMethod() {
1510        …
1511    }
```

## 9.11 @Integrity

1513 The following Java code defines the **@Integrity** annotation:

1514
```
1515    package org.oasisopen.sca.annotation;
1516
1517    import static java.lang.annotation.ElementType.FIELD;
1518    import static java.lang.annotation.ElementType.METHOD;
1519    import static java.lang.annotation.ElementType.PARAMETER;
1520    import static java.lang.annotation.ElementType.TYPE;
1521    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1522    import static org.oasisopen.Constants.SCA_PREFIX;
1523
1524    import java.lang.annotation.Inherited;
1525    import java.lang.annotation.Retention;
1526    import java.lang.annotation.Target;
1527
1528    @Inherited
1529    @Target({TYPE, FIELD, METHOD, PARAMETER})
1530    @Retention(RUNTIME)
1531    @Intent(Integrity.INTEGRITY)
1532    public @interface Integrity {
1533        String INTEGRITY = SCA_PREFIX + "integrity";
1534        String INTEGRITY_MESSAGE = INTEGRITY + ".message";
1535        String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";
1536
1537        /**
1538         * List of integrity qualifiers (such as "message" or "transport").
1539         *
1540         * @return integrity qualifiers
1541         */
1542        @Qualifier
1543        String[] value() default "";
1544    }
```

1546 The **@Integrity** annotation is used to indicate that the invocation requires integrity (ie no
1547 tampering of the messages between client and service).

1548 See the section on Application of Intent Annotations for samples and details.

## 9.12 @Intent

1550 The following Java code defines the **@Intent** annotation:

1551
1552    package org.osoa.sca.annotation;

1553
1554    import static java.lang.annotation.ElementType.ANNOTATION_TYPE;

```
1555    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1556    import java.lang.annotation.Retention;
1557    import java.lang.annotation.Target;
1558
1559    @Target({ANNOTATION_TYPE})
1560    @Retention(RUNTIME)
1561    public @interface Intent {
1562       /**
1563        * The qualified name of the intent, in the form defined by
1564        * {@link javax.xml.namespace.QName#toString}.
1565        * @return the qualified name of the intent
1566        */
1567       String value() default "";
1568
1569       /**
1570        * The XML namespace for the intent.
1571        * @return the XML namespace for the intent
1572        */
1573       String targetNamespace() default "";
1574
1575       /**
1576        * The name of the intent within its namespace.
1577        * @return name of the intent within its namespace
1578        */
1579       String localPart() default "";
1580    }
```

1581

1582    The @Intent annotation is used for the creation of new annotations for specific intents.  It is not
1583    expected that the @Intent annotation will be used in application code.

1584    See the section "How to Create Specific Intent Annotations" for details and samples of how to
1585    define new intent annotations.

## 8.99.13 @OneWay

1587    The following Java code defines the **@OneWay** annotation:

1588

```
1589    package org.oasisopen.sca.annotation;
1590
1591    import static java.lang.annotation.ElementType.METHOD;
1592    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1593    import java.lang.annotation.Retention;
1594    import java.lang.annotation.Target;
1595
1596    @Target(METHOD)
1597    @Retention(RUNTIME)
1598    public @interface OneWay {
1599
1600
1601    }
```

1602

1603    The @OneWay annotation is used on a Java interface or class method to indicate that invocations
1604    will be dispatched in a non-blocking fashion as described in the section on Asynchronous
1605    Programming.

1606    The @OneWay annotation has no attributes.

1607    The following snippet shows the use of the @OneWay annotation on an interface.

```
1608    package services.hello;
1609
1610    import org.oasisopen.sca.annotation.OneWay;
1611
1612    public interface HelloService {
1613        @OneWay
1614        void hello(String name);
1615    }
```

## 9.14 @PolicySet

1617 The following Java code defines the **@PolicySets** annotation:

1618
```
1619    package org.oasisopen.sca.annotation;
1620
1621    import static java.lang.annotation.ElementType.FIELD;
1622    import static java.lang.annotation.ElementType.METHOD;
1623    import static java.lang.annotation.ElementType.PARAMETER;
1624    import static java.lang.annotation.ElementType.TYPE;
1625    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1626
1627    import java.lang.annotation.Retention;
1628    import java.lang.annotation.Target;
1629
1630    @Target({TYPE, FIELD, METHOD, PARAMETER})
1631    @Retention(RUNTIME)
1632    public @interface PolicySets {
1633        /**
1634         * Returns the policy sets to be applied.
1635         *
1636         * @return the policy sets to be applied
1637         */
1638        String[] value() default "";
1639    }
```

1640
1641 The **@PolicySet** annotation is used to attach an SCA Policy Set to a Java implementation class or
1642 to one of its subelements.

1643 See the section "Policy Set Annotations" for details and samples.

## 8.109.15 @Property

1645 The following Java code defines the **@Property** annotation:

```
1646    package org.oasisopen.sca.annotation;
1647
1648    import static java.lang.annotation.ElementType.METHOD;
1649    import static java.lang.annotation.ElementType.FIELD;
1650    import static java.lang.annotation.ElementType.PARAMETER;
1651    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1652    import java.lang.annotation.Retention;
1653    import java.lang.annotation.Target;
1654
1655    @Target({METHOD, FIELD, PARAMETER})
1656    @Retention(RUNTIME)
1657    public @interface Property {
1658
1659        String name() default "";
```

```
1660          boolean required() default true;
1661      }
1662
```

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

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

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

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

The @Property annotation has the following attributes:

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

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

The following snippet shows a property field definition sample.

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

The following snippet shows a property setter sample

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

    ....

}
```

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

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

```
...
private List<String> helloConfigurationProperty;

@Property(required=true)
public void setHelloConfigurationProperty(List<String> property) {
        helloConfigurationProperty = property;
}
```

1708    ...

## 9.16 @Qualifier

1710 The following Java code defines the **@Qualifier** annotation:

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

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

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

@Target(METHOD)
@Retention(RUNTIME)
public @interface Qualifier {
}
```

1725 The @Qualifier annotation is applied to an attribute of an intent annotation definition, defined
1726 using the @Intent annotation, to indicate that the attribute provides qualifiers for the intent. The
1727 @Qualifier annotation MUST be used in an intent annotation definition where the intent has
1728 qualifiers.

1729 See the section "How to Create Specific Intent Annotations" for details and samples of how to
1730 define new intent annotations.

## 8.119.17 @Reference

1732 The following Java code defines the **@Reference** annotation:

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

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
@Target({METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Reference {

    String name() default "";
    boolean required() default true;
}
```

1750 The @Reference annotation type is used to annotate a Java class field, a setter method, or a
1751 constructor parameter that is used to inject a service that resolves the reference. The interface of
1752 the service injected is defined by the type of the Java class field or the type of the input parameter
1753 of the setter method or constructor.

1754 The @Reference annotation MUST NOT be used on a class field that is declared as final.

1755 References maycan also be injected via setter methods even when the @Reference annotation is
1756 not present.  However, the @Reference annotation must be used in order to inject a reference
1757 onto a non-public field.  In the case where there is no @Reference annotation, the name of the
1758 reference is the same as the name of the field or setter.

1759    Where there is both a setter method and a field for a reference, the setter method is used.

1760    The @Reference annotation has the following attributes:

1761    • *name (optional)* – the name of the reference. For a field annotation, the default is the
1762      name of the field of the Java class.  For a setter method annotation, the default is the
1763      JavaBeans property name corresponding to the setter method name.  For a constructor
1764      parameter annotation, there is no default and the name attribute MUST be present.

1765    • *required (optional)* – whether injection of service or services is required. Defaults to true.
1766      For a constructor parameter annotation, this attribute MUST have the value true.

1767

1768    The following snippet shows a reference field definition sample.

1769

```
1770    @Reference(name="stockQuote", required=true)
1771    protected StockQuoteService stockQuote;
```

1772

1773    The following snippet shows a reference setter sample

1774

```
1775    @Reference(name="stockQuote", required=true)
1776    public void setStockQuote( StockQuoteService theSQService ) {

1777         ...

1778    }
```

1779

1780    The following fragment from a component implementation shows a sample of a service reference
1781    using the @Reference annotation. The name of the reference is "helloService" and its type is
1782    HelloService. The clientMethod() calls the "hello" operation of the service referenced by the
1783    helloService reference.

1784

```
1785    package services.hello;

1786
1787    private HelloService helloService;

1788
1789    @Reference(name="helloService", required=true)
1790    public setHelloService(HelloService service) {
1791            helloService = service;
1792    }

1793
1794    public void clientMethod() {
1795            String result = helloService.hello("Hello World!");
1796            …
1797    }
```

1798

1799    The presence of a @Reference annotation is reflected in the componentType information that the
1800    runtime generates through reflection on the implementation class.  The following snippet shows
1801    the component type for the above component implementation fragment.

1802

```
1803    <?xml version="1.0" encoding="ASCII"?>
1804    <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">

1805
1806        <!-- Any services offered by the component would be listed here -->
```

```
1807        <reference name="helloService" multiplicity="1..1">
1808              <interface.java interface="services.hello.HelloService"/>
1809        </reference>
1810
1811    </componentType>
1812
```

1813 If the reference is not an array or collection, then the implied component type has a reference
1814 with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference *required*
1815 attribute – 1..1 applies if required=true.

1816

1817 If the reference is defined as an array or as any type that extends or implements *java.util.Collection*,
1818 then the implied component type has a reference with a *multiplicity* of either *1..n or 0..n*, depending
1819 on whether the *required* attribute of the **@Reference** annotation is set to true or false – 1..n applies if
1820 required=true.

1821

1822 The following fragment from a component implementation shows a sample of a service reference
1823 definition using the @Reference annotation on a java.util.List. The name of the reference is
1824 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
1825 services referenced by the helloServices reference.  In this case, at least one HelloService should
1826 be present, so *required* is true.

```
1827
1828        @Reference(name="helloServices", required=true)
1829        protected List<HelloService> helloServices;
1830
1831        public void clientMethod() {
1832
1833              …
1834              for (int index = 0; index < helloServices.size(); index++) {
1835                    HelloService helloService =
1836                          (HelloService)helloServices.get(index);
1837                    String result = helloService.hello("Hello World!");
1838              }
1839              …
1840        }
1841
```

1842 The following snippet shows the XML representation of the component type reflected from for the
1843 former component implementation fragment. There is no need to author this component type in
1844 this case since it can be reflected from the Java class.

1845

```
1846    <?xml version="1.0" encoding="ASCII"?>
1847    <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1848
1849        <!-- Any services offered by the component would be listed here -->
1850        <reference name="helloServices" multiplicity="1..n">
1851              <interface.java interface="services.hello.HelloService"/>
1852        </reference>
1853
1854    </componentType>
1855
```

1856 At runtime, the representation of an unwired reference depends on the reference's multiplicity. An
1857 unwired reference with a multiplicity of 0..1 must be null. An unwired reference with a multiplicity
1858 of 0..N must be an empty array or collection.

### 8.11.19.17.1 Reinjection

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

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

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

If an operation is called on a reference where the target of that reference has been undeployed, the SCA runtime SHOULD throw InvalidServiceException. If an operation is called on a reference where the target of the reference has become unavailable for some reason, the SCA runtime SHOULD throw ServiceUnavailableException. If the target of the reference is changed, the reference 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.

A ServiceReference that has been obtained from a reference by ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast(). If the reference is subsequently reinjected, the ServiceReference obtained from the original reference MUST continue to work as if the reference target was not changed. If the target of a ServiceReference has been undeployed, the SCA runtime SHOULD throw InvalidServiceException when an operation is invoked on the ServiceReference. If the target of a ServiceReference has become unavailable, the SCA runtime SHOULD throw ServiceUnavailableException when an operation is invoked on the ServiceReference. If the target of a ServiceReference is changed, the reference 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.

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

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

|  | **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. |

\* Other conditions:

1. The component MUST NOT be STATELESS scoped.

2. The reference MUST use either field-based injection or setter injection. References that are injected through constructor injection MUST NOT be changed.

\*\* Result of invoking ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().

1898

## 8.129.18 @Remotable

1899

1900 The following Java code defines the **@Remotable** annotation:

1901

```
1902    package org.oasisopen.sca.annotation;
1903
1904    import static java.lang.annotation.ElementType.TYPE;
1905    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1906    import java.lang.annotation.Retention;
1907    import java.lang.annotation.Target;
1908
1909
1910    @Target(TYPE)
1911    @Retention(RUNTIME)
1912    public @interface Remotable {
1913
1914    }
1915
```

1916 The @Remotable annotation is used to specify a Java service interface as remotable. A remotable
1917 service can be published externally as a service and must be translatable into a WSDL portType.

1918 The @Remotable annotation has no attributes.

1919 The following snippet shows the Java interface for a remotable service with its @Remotable
1920 annotation.

```
1921    package services.hello;
1922
1923    import org.oasisopen.sca.annotation.*;
1924
1925    @Remotable
1926    public interface HelloService {
1927
1928        String hello(String message);
```

```
1929        }
1930
```

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

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

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

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

The following snippet shows a remotable Java service interface.

```
1945
1946        package services.hello;
1947
1948        import org.oasisopen.sca.annotation.*;
1949
1950        @Remotable
1951        public interface HelloService {
1952
1953            String hello(String message);
1954        }
1955
1956        package services.hello;
1957
1958        import org.oasisopen.sca.annotation.*;
1959
1960        @Service(HelloService.class)
1961        public class HelloServiceImpl implements HelloService {
1962
1963            public String hello(String message) {
1964                    ...
1965            }
1966        }
```

## 9.19 @Requires

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

```
1969
1970        package org.oasisopen.sca.annotation;
1971
1972        import static java.lang.annotation.ElementType.FIELD;
1973        import static java.lang.annotation.ElementType.METHOD;
1974        import static java.lang.annotation.ElementType.PARAMETER;
1975        import static java.lang.annotation.ElementType.TYPE;
1976        import static java.lang.annotation.RetentionPolicy.RUNTIME;
1977
1978        import java.lang.annotation.Inherited;
1979        import java.lang.annotation.Retention;
1980        import java.lang.annotation.Target;
```

```
1981
1982        @Inherited
1983        @Retention(RUNTIME)
1984        @Target({TYPE, METHOD, FIELD, PARAMETER})
1985        public @interface Requires {
1986            /**
1987             * Returns the attached intents.
1988             *
1989             * @return the attached intents
1990             */
1991            String[] value() default "";
1992        }
1993
```

1994    The **_@Requires_** annotation supports general purpose intents specified as strings. User can also
1995    define specific intents using @Intent annotation.

1996    See the section "General Intent Annotations" for details and samples.

### 8.139.20 @Scope

1998    The following Java code defines the **_@Scope_** annotation:

```
1999        package org.oasisopen.sca.annotation;
2000
2001        import static java.lang.annotation.ElementType.TYPE;
2002        import static java.lang.annotation.RetentionPolicy.RUNTIME;
2003        import java.lang.annotation.Retention;
2004        import java.lang.annotation.Target;
2005
2006        @Target(TYPE)
2007        @Retention(RUNTIME)
2008        public @interface Scope {
2009
2010            String value() default "STATELESS";
2011        }
```

2012    The @Scope annotation mayMUST only be used on a service's implementation class. It is an error
2013    to use this annotation on an interface.

2014    The @Scope annotation has the following attribute:

2015        •   **_value_** – the name of the scope.
2016            For 'STATELESS' implementations, a different implementation instance maycan be used to
2017            service each request. Implementation instances maycan be newly created or be drawn
2018            from a pool of instances.
2019            SCA defines the following scope names, but others can be defined by particular Java-
2020            based implementation types:
2021            STATELESS
2022            COMPOSITE

2023    The default value is STATELESS.

2024    The following snippet shows a sample for a COMPOSITE scoped service implementation:

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

```
2034              ...
2035          }
2036      }
2037
```

### 8.149.21 @Service

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

```
2040    package org.oasisopen.sca.annotation;
2041
2042    import static java.lang.annotation.ElementType.TYPE;
2043    import static java.lang.annotation.RetentionPolicy.RUNTIME;
2044    import java.lang.annotation.Retention;
2045    import java.lang.annotation.Target;
2046
2047    @Target(TYPE)
2048    @Retention(RUNTIME)
2049    public @interface Service {
2050
2051        Class<?>[] interfaces() default {};
2052        Class<?> value() default Void.class;
2053    }
2054
```

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.

The @Service annotation has the following attributes:

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

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

Only one of these attributes should be specified.

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

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

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.

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

```
2076    package services.hello;
2077
2078    import org.oasisopen.sca.annotation.Service;
2079
2080    @Service(HelloService.class)
2081    public class HelloServiceImpl implements HelloService {
2082
2083        public void hello(String name) {
```

```
2084            System.out.println("Hello " + name);
2085        }
2086    }
2087
```

## 9.22 Security Implementation Policy Annotations

JSR 250 "Common Annotations for the Java Platform" defines the following annotations that can be used for implementation security policy:

javax.annotation.security.RunAs
javax.annotation.security.RolesAllowed
javax.annotation.security.PermitAll
javax.annotation.security.DenyAll
javax.annotation.security.DeclareRoles

Based on JSR 250, the RunAs , DeclareRoles annotations can be specified on a class; the RolesAllowed , PermitAll , DenyAll annotations can be specified on a class or on method(s).

Please check JSR250 and SCA Policy spec for details on the meaning of these annotations . Please check the section [10.6.2] Security Implementation Policy for the details on how these annotations are mapped into Policy framework.

## 910 WSDL to Java and Java to WSDL

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.

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.

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.

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

- No support for holders

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

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

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.

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.

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

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

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

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

- a. a method name that is M's method name with the characters "Async" appended, and
- 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
- c. a return type of Future<?>

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

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

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

```
2150    <part name="ticker" type="xsd:string"/>
2151  </message>
2152
2153  <message name="getPriceResponse">
2154    <part name="price" type="xsd:float"/>
2155  </message>
2156
2157  <portType name="StockQuote">
2158   <operation name="getPrice">
2159      <input message="tns:getPrice"/>
2160      <output message="tns:getPriceResponse"/>
2161   </operation>
2162  </portType>
```

2163

2164 The JAX-WS asynchronous mapping will produce the following Java interface:

```
2165   // asynchronous mapping
2166   @WebService
2167   public interface StockQuote {
2168    float getPrice(String ticker);
2169    Response<Float> getPriceAsync(String ticker);
2170    Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
2171   }
```

2172

2173 For SCA interface definition purposes, this is treated as equivalent to the following:

```
2174   // synchronous mapping
2175   @WebService
2176   public interface StockQuote {
2177    float getPrice(String ticker);
2178   }
```

2179

2180 SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above
2181 example, if the client implementation uses the asynchronous form of the interface, the two
2182 additional getPriceAsync() methods can be used for polling and callbacks as defined by the JAX-
2183 WS specification.

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

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
        elementFormDefault="qualified">

    <include schemaLocation="sca-core.xsd"/>

    <element name="interface.java" type="sca:JavaInterface"
                substitutionGroup="sca:interface"/>
    <complexType name="JavaInterface">
        <complexContent>
            <extension base="sca:Interface">
                <sequence>
                    <any namespace="##other" processContents="lax"
                        minOccurs="0" maxOccurs="unbounded"/>
                </sequence>
                <attribute name="interface" type="NCName" use="required"/>
                <attribute name="callbackInterface" type="NCName"
                    use="optional"/>
                <anyAttribute namespace="##any" processContents="lax"/>
            </extension>
        </complexContent>
    </complexType>
</schema>
```

# B. Conformance Items

2212 This section contains a list of conformance items for the SCA Java Common Annotations and APIs
2213 specification.
2214

| 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. |

2215

# C. Acknowledgements

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

**Participants:**

[Participant Name, Affiliation | Individual Member]

[Participant Name, Affiliation | Individual Member]

# D. Non-Normative Text

# E. Revision History

2225 [optional; should not be included in OASIS Standards]

2226

| 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 Combellack | 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 |

2227

| **Page 26: [1] Formatted** | **Yang Lei** | **2/3/2009 1:16:00 PM** |

Normal, Indent: Left: 0.75", No bullets or numbering, Don't adjust space between Latin and Asian text

| **Page 26: [2] Formatted** | **Yang Lei** | **2/6/2009 1:39:00 PM** |

Normal, No bullets or numbering, Don't adjust space between Latin and Asian text

| **Page 26: [3] Formatted** | **Yang Lei** | **2/3/2009 1:16:00 PM** |

Normal, Indent: Left: 0.75", No bullets or numbering, Don't adjust space between Latin and Asian text

| **Page 26: [4] Formatted** | **Yang Lei** | **2/6/2009 1:39:00 PM** |

Normal, No bullets or numbering, Don't adjust space between Latin and Asian text

| **Page 26: [5] Formatted** | **Yang Lei** | **2/3/2009 1:16:00 PM** |

Normal, Indent: Left: 0.75", No bullets or numbering, Don't adjust space between Latin and Asian text

| **Page 26: [6] Formatted** | **Yang Lei** | **2/6/2009 1:42:00 PM** |

Normal, Don't adjust space between Latin and Asian text

| **Page 26: [7] Formatted** | **Yang Lei** | **2/3/2009 1:18:00 PM** |

Normal, Indent: Left: 0.75", No bullets or numbering, Don't adjust space between Latin and Asian text

| **Page 26: [8] Formatted** | **Yang Lei** | **2/6/2009 1:42:00 PM** |

Normal, No bullets or numbering, Don't adjust space between Latin and Asian text

| **Page 26: [9] Formatted** | **Yang Lei** | **2/3/2009 1:18:00 PM** |

Normal, Indent: Left: 0.75", No bullets or numbering, Don't adjust space between Latin and Asian text