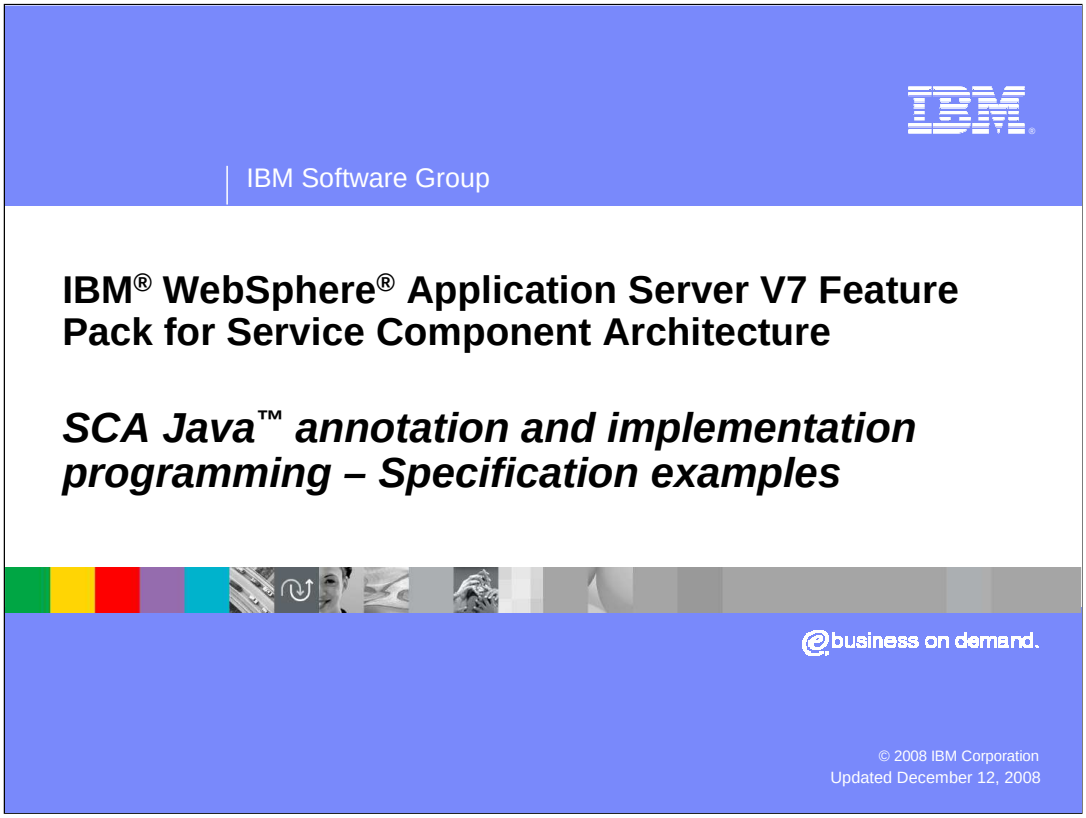
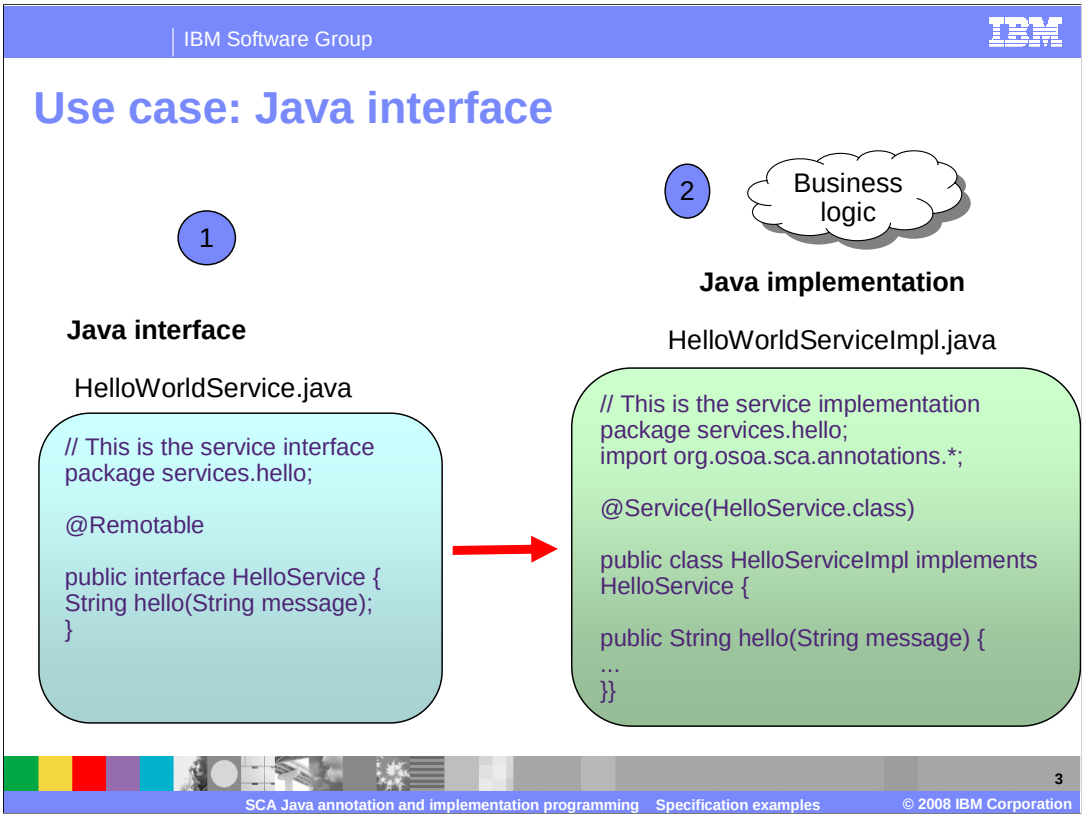


This presentation will cover the examples from the two specifications for SCA V1.0 programming model – Java annotations and Java implementation.

Section

Java specifications examples

This section will discuss the Java specifications with SCA examples where applicable.



The first one is a simple Java interface use case.

Starting with a Java interface, you would then write Java implementation with its business logic meaning create a POJO implementation/interface relationship as shown.

Note the @Service annotation to refer back to Java interface in this case service interface.

Also note the @Remotable annotation which will be discussed more in the next slide

Local and remotable services

- A Java service interface may use @Remotable to declare that a service can be accessed remotely
- If @Remotable is not used, the service is considered local

```
package services.hello;

@Remotable

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


A Java service contract defined by an interface or implementation class may use @Remotable to declare that the service follows the semantics of remotable services as defined by the SCA Assembly Specification. The above example demonstrates the use of @Remotable annotation.

If @Remotable is not used, then the service is assumed to be local.

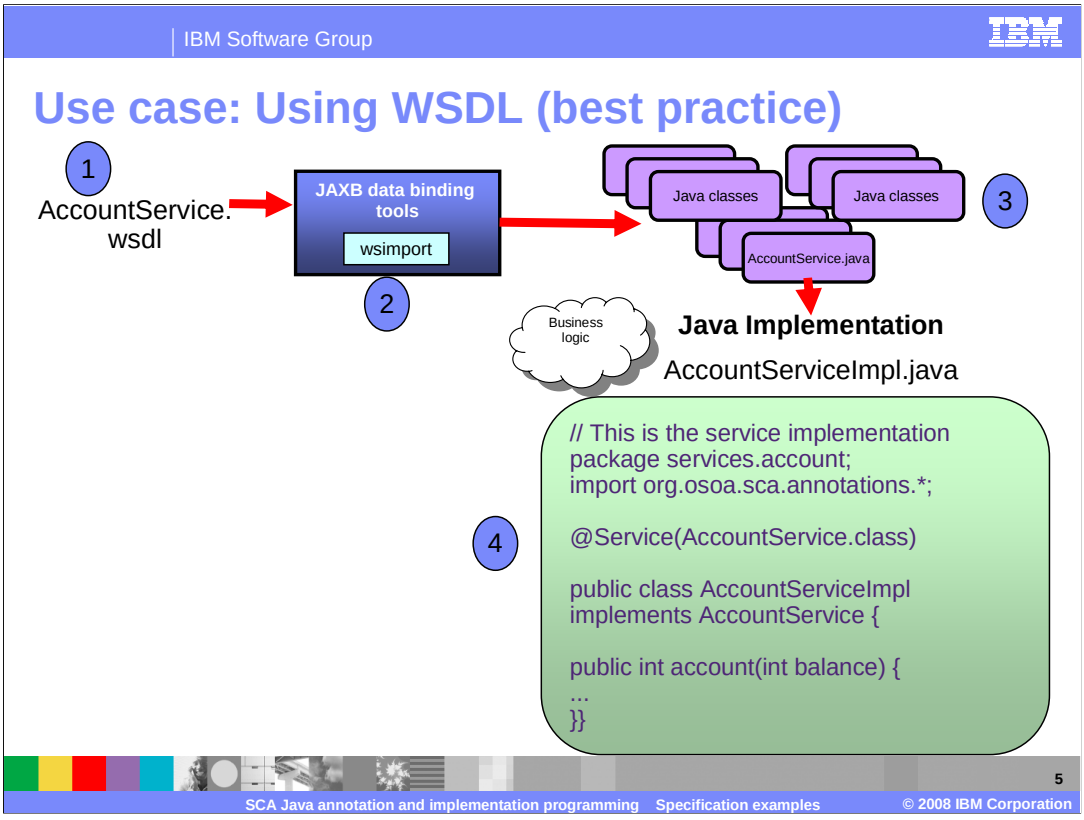
If an implementation class has implemented interfaces that are not decorated with an @Remotable

annotation, the class is considered to implement a single local service whose type is defined by the class

Note that the style of local interfaces is typically ***fine grained*** and 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.

Also note that a remote service could be running in a different process on the same physical computer or on a different computer



Here is an example of SCA feature pack best practice use case where you start from WSDL interface for either an implementation or client.

In this scenario, you start with WSDL interface (portType) plus XSDs example AccountService.wsdl. There after, run codegen (wsimport) and generate the Java classes such as AccountService.java. Finally write Java implementation (AccountServiceImpl.java) with its business logic . The Java implementation implement the generated SEI (this is the one with @WebService) in the POJO sense) and add @Service to refer back to generated SEI class. Finally you can expose Account Service over Web services bindings.

Annotation scope

- SCA runtime can manage the state of a component as specified by the @Scope annotation

```
@Scope("stateless")
pubic interface ShoppingCartService {
   public void addToCart(int quantity);
}
```

Added to the interface or the implementation

- Supported scopes are:
  - stateless, composite
- Implementations can include life cycle methods
  - @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 life cycle 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

The @Scope annotation type is used on either a service's interface definition or on a service implementation class itself.

This annotation can be added to either the interface or implementation class definition. The possible values for the @Scope annotation are: **stateless, session, conversation**, and **composite**. However, SCA feature pack only supports **stateless** and **composite** scopes.

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

An implementation type may allow component implementations to declare ***life cycle methods*** that are called when an implementation is instantiated or the scope is expired. ***@Init*** denotes the method to be called upon first use of an instance during the lifetime of the scope. ***@Destroy*** specifies the method to be called when the scope ends. Note that only public, no argument methods may be annotated as life cycle methods.

Note that scope impacts the threading of the programming model that is stateless=single-threaded composite=multi-threaded.

Supported security annotations

- @DeclareRoles (Servlet 2.5 and EJB 3)
- @RunAs (Servlet 2.5 and EJB 3)
  - role name used in the RunAs annotation must be defined in the deployment descriptor.
- @DenyAll (EJB 3 only)
- @PermitAll (EJB 3 only)
- @RolesAllowed (EJB 3 only)

In Java EE 5, The security roles and policies can be defined using annotations within the deployment descriptor. During the installation of the application, the security policies and roles defined using annotations are merged with the security policies and roles defined within the deployment descriptor.

The list shown shows the supported security annotations.

Note that the role name used in the RunAs annotation must be defined in the deployment descriptor.

Policy annotations : Overview

- SCA provides policies that influence how implementations, services and references behave at runtime
- Policy annotations include:
  - General intent annotations : @Requires
    - Example: @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
  - Specific intent annotations
    - Example: @Authentication({“message”, “transport”})
- Policy annotations implementation example

```
xmlns:qos="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
  <binding.ws qos:wsPolicySet="WSHTTPS default"/>
```


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 include Intents and Policy Sets, where intents express abstract, high-level policy requirements and policy sets express low-level detailed concrete policies.

General intent annotations:

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.

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

@Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})

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

Specific intent annotations - Java annotations that correspond to specific policy intents

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

@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 example shown shows the SCA feature pack specific PolicySet implementation with Web services binding.

Configuration properties

- Properties are used to allow data configuration for a service component definition
- Configuration properties are identified by
  - @Property annotation
- @Property can be used on
  - Public and protected data members
  - Setter methods
  - Constructor

The @Property annotation type is used to annotate a Java class field or a setter method 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 setter method input argument.

The @Property annotation may be used on protected or public fields and on setter methods or on a constructor method.

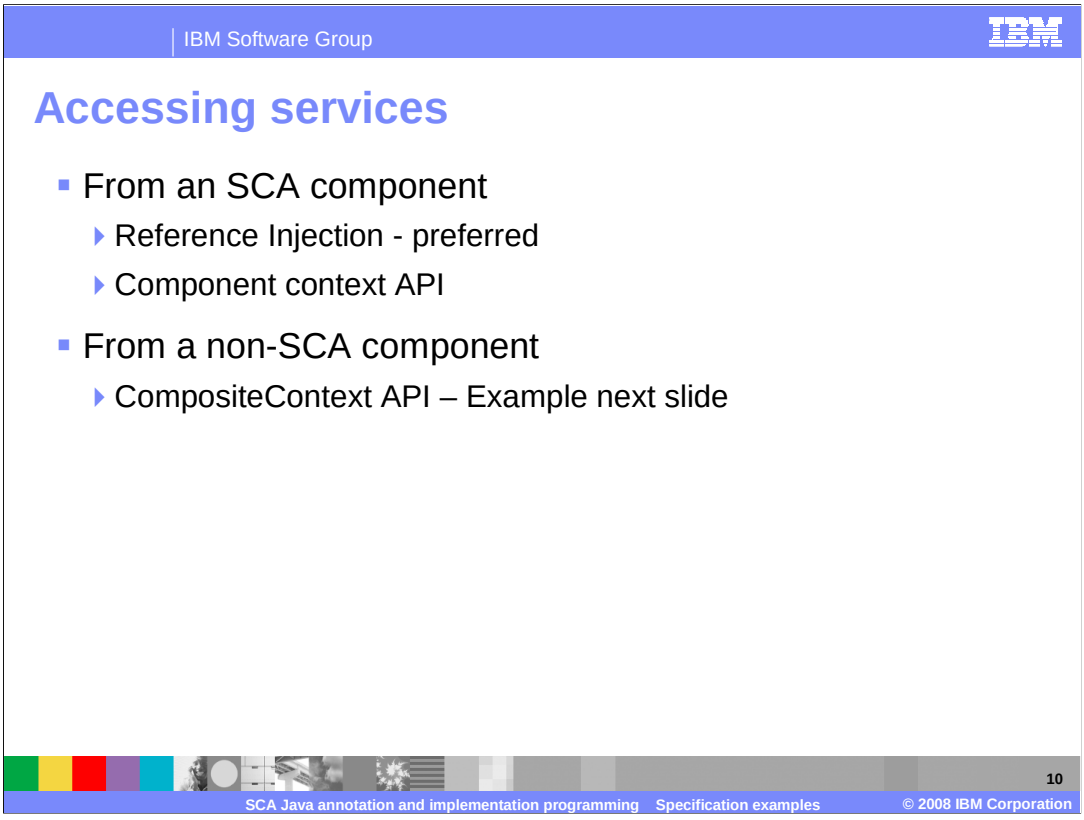
Properties may also be injected through public setter methods even when the @Property annotation is not present. However, the @Property annotation must be used in order to inject a property onto a non-public field. In the case where there is no @Property annotation, the name of the property is the same 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 **name** and **required** optional attributes:

• ***name (optional)*** – the name of the property, defaults to the name of the field of the Java class

• ***required (optional)*** – specifies whether injection is required, defaults to false.



An SCA component may obtain a service reference through injection or programmatically through the

component Context API. Using reference injection is the recommended way to access a service, since it results in code with minimal use of middleware APIs. The ComponentContext API should be used in cases where reference injection is not possible.

For Non-SCA client (e.g the typical JSP of the samples provided) it gets a reference to some component service so it can invoke this service. It does this using the CompositeContext API which returns a reference to a top-level component service deployed to the domain. This reference is a proxy to the service which you can then invoke.

The next slide demonstrates the use of the CompositeContext API by non-SCA code.

Note that for this release, the limitation is that the invocation always happens over default binding. getService is not supported for binding.ws (Web services binding).

Non-SCA client: Example

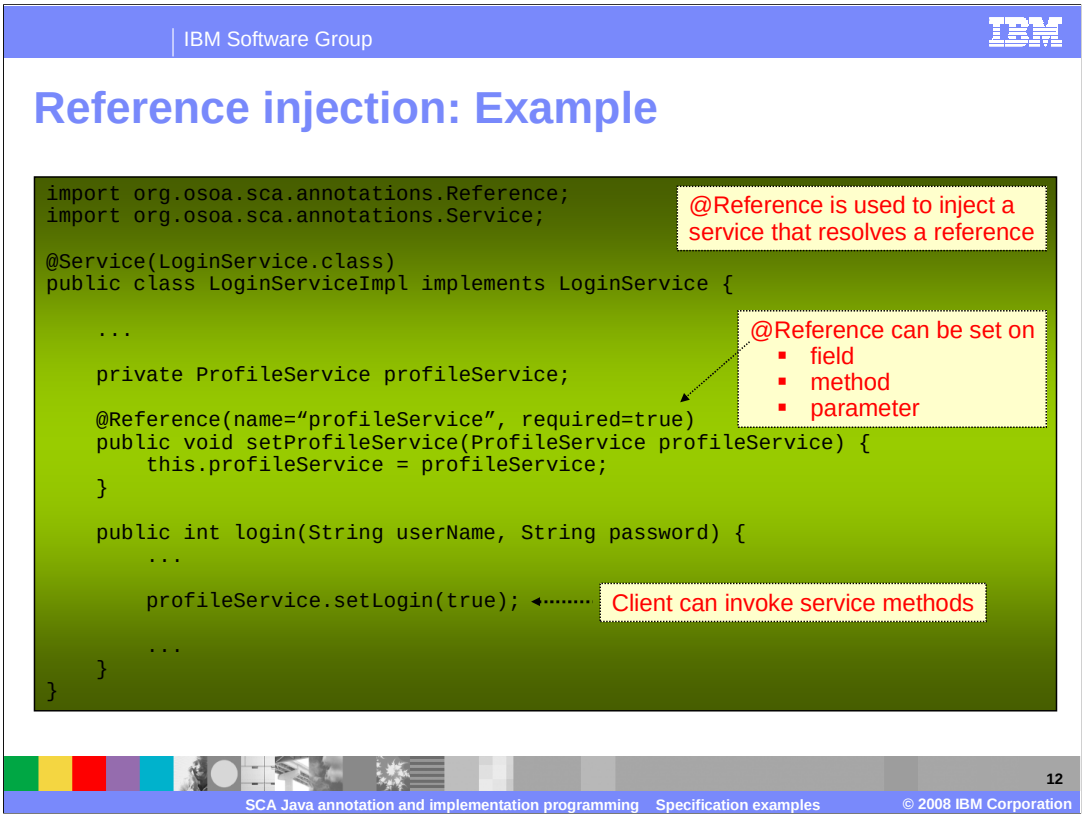
```
public class HelloServiceClient {
    public static void main(String[] args) throws Exception {

        // Locate the service using SCA APIs
        CompositeContext context = CurrentCompositeContext.getContext();
        HelloService helloService = context.locateService(HelloService.class,
        "HelloComponent");
        ...
        // Invoke the service
        String result = helloService.hello("Hello World");
    }
}
```


This is an example of a non-SCA client for a single-service domain component, **HelloComponent** with service **HelloService**

The SCA API class CompositeContext allows a service to be located by name. This is the component name as it appears in the composite file. The result of this location step is a service instance or proxy that implements the service interface. When a method is invoked by the client code (hello() in this case), the SCA runtime will dispatch the operation parameters to the correct method. The method reference would be a method in the component implementation according to the definitions in the SCDL file.

Note that: Invocation of located services occurs only over **binding.sca**. It is not supported for binding.ws.



This is an example of a @Reference annotation.

The @Reference annotation type is used to annotate a Java class field or a setter method that is used to have a service reference injected. The interface of the service injected is defined by the type of the Java class field or the type of the setter method input argument.

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 an ***@Reference*** annotation.

References may also be injected through public setter methods even when the @Reference annotation is not present. However, the @Reference annotation must be used in order to inject a reference onto a non public field. In the case where there is no @Reference annotation, the name of the reference is the same as the name of the field or setter. Where there is both a setter method and a field for a reference, the setter method is used.

****

private HelloService helloService;

@Reference (name="helloService", required=**true**)

public setHelloService(HelloService service){

helloService = service;

}

public void clientMethod() {

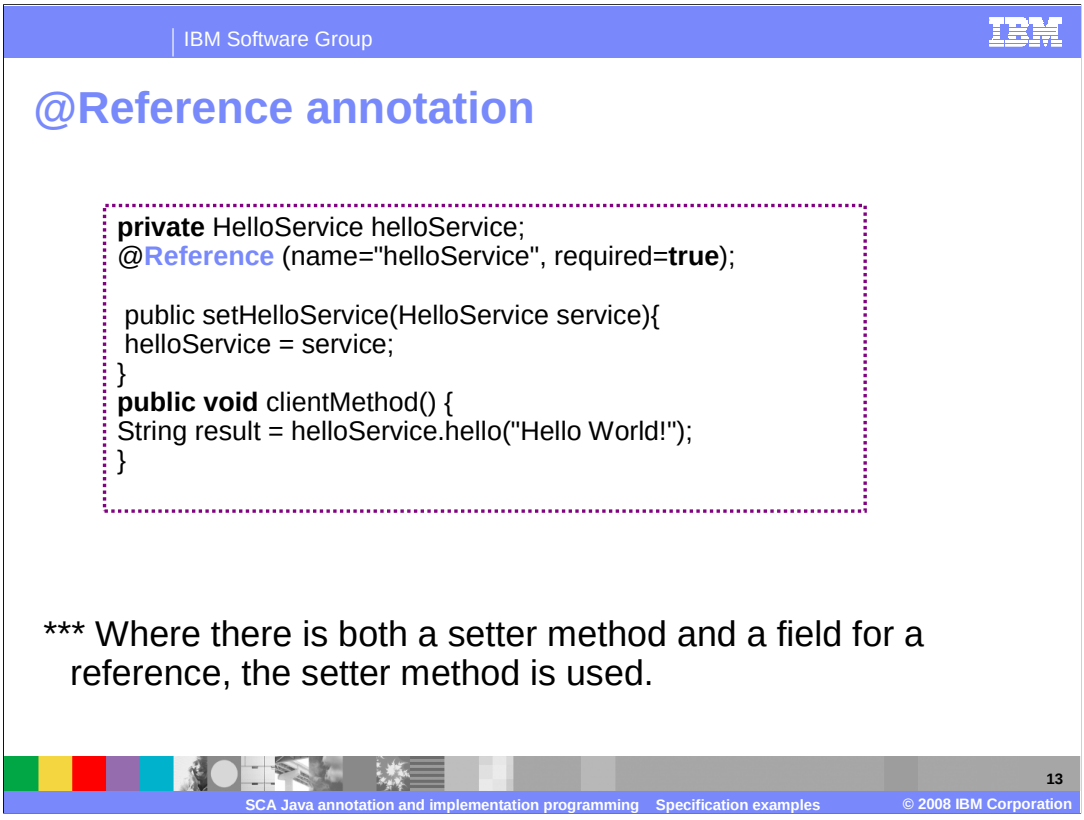
String result = helloService.hello("Hello World!");

}

Like the Property annotation, The @Reference annotation has these attributes:

• ***name (optional)*** – the name of the reference, defaults to the name of the field of the Java class

• ***required (optional)*** – whether injection of service or services is required. Defaults to true.

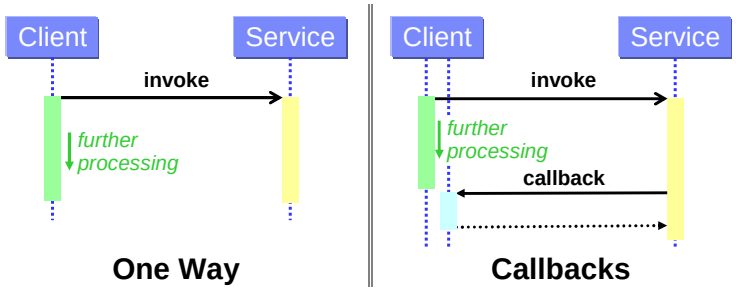


This is another example of reference annotation.

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

Asynchronous model

- SCA provides the ability for services to be called synchronously or asynchronously
- Asynchronous support
  - One way (non-blocking)
  - Callbacks



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, and callbacks.

The Java annotations specification includes support for both synchronous and asynchronous invocation styles. The examples shown so far illustrate a synchronous interaction style.

Asynchronous model: One way

- Simplest type of asynchronous invocation
  - Client invokes a service and continues processing without waiting for the service to complete
- Methods supporting a one way invocation style
  - Are identified with the @OneWay annotation
  - Must return void
  - Must not throw any exceptions

@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 that returns "void" and has no declared exceptions may be marked with an @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.

Note that SCA does not currently define a mechanism for making non-blocking calls to methods that return values or are declared to throw exceptions. It is recommended that you define one-way methods as often as possible, in order to give the greatest degree of binding flexibility.

Asynchronous model: Callbacks

- Provides communication from a service provider back to the client through a callback service
  - Also known as a bi-directional services
- Services have two interfaces
  - Interface for the provided service
  - Interface for the callback
- Clients provide an implementation for the callback interface
- The @Callback annotation is used to specify a callback interface

The other asynchronous interaction style is the callback style. In this type of interaction style the client invokes a service method, and control is immediately returned to the client while the service processes the request. The client is notified by the service provider by way of the callback interface. The @Callback annotation is used to specify the callback interface used for a particular interface.

With this type of asynchronous interaction, the client must implement a specific callback interface that is used by the service provider to handle the response to an asynchronous call.

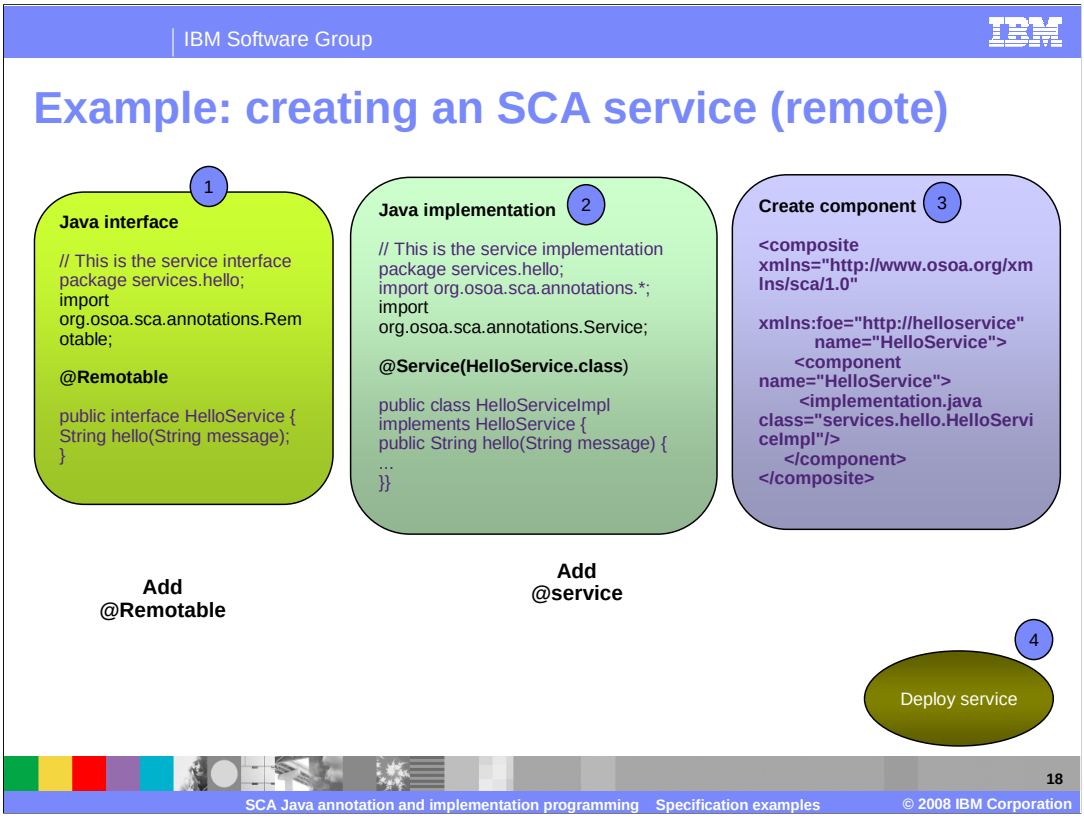
Limitations

- Only stateless and composite are valid values for the @Scope annotation
- @conversational services are not supported

Here is a list of some limitations on the Java implementation and annotation model as far as this release is concerned:

Only stateless and composite are valid values for the @Scope annotation

@conversational services are not supported.



This example will walk you through quick steps for creating an SCA service in terms of Java implementation/Annotations model.

Start with Java interface example HelloService. If you are creating a service with a remotable interface, add the @Remotable annotation. Unless you have an existing Java implementation, write a Java implementation of the generated Java interface that reflects your business logic. Be sure to add the @Service annotation to the Java implementation. Create a component using the component implementation. You will create a component definition in a SCDL file that references the original Java implementation class, including its Java interface. In SCA, a component is a configured instance of a component implementation. There are other aspects of defining a component that are not shown here such as configuring bindings, configuring property values, defining intents, attaching policy sets, and resolving references. This is a simple general example. After your component is defined as part of a deployable composite, either directly or recursively you are ready to deploy the SCA service by creating an SCA business level application.

Section

Summary and references

Finally a summary.

Summary

- The WebSphere Application Server V7 Feature Pack for SCA provides support for Java component implementation model and Java common annotations
- Specification describes annotations and programming support for key SCA features

The Java experience for SCA programmers should be one that leverages the ease-of-use characteristics of Java 5, which uses annotations and dependency injection. It demonstrates a real, tangible separation of concerns that frees the business logic from configuration and protocol specific APIs that burden applications today. When this Java experience is coupled with the visual composition metaphors that can be realized by tools to edit the SCA assembly. The focus of developing the application aligns directly with the principles of SOA – loosely-coupled, reused, coarse-grained services composed from a broad-array of implementation and implementation technologies. That focus squarely on the Business Service itself, and not burdened with specific implementation.

Reference

- SCA Java annotations and APIs

http://www.osoa.org/download/attachments/35/SCA_JavaAnnotationsAndAPIs_V100.pdf?version=1

- SCA Java component implementation

http://www.osoa.org/download/attachments/35/SCA_JavaComponentImplementation_V100.pdf?version=1

Here is a reference to the specs.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WASv7SCA_JavaProgrammingModel_specifications.ppt

This module is also available in PDF format at:
../WASv7SCA_JavaProgrammingModel_specifications.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers

IBM, the IBM logo, ibm.com, and the following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

WebSphere

If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of other IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

EJB, Java, JSP, and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.