Basic DOF Operations

Java Programmer's Guide

Version 7.0

Table of Contents

Chapter 1: Introduction 1

How to Read This Guide 1

The Time-Based Alarm Interface 1

Description of Interface Items 2

Item Identifiers 2

Interface Code 3

Chapter 2: Basic Setup 5

Obtaining DOFSystem Instances 5

Creating DOFObjectID Instances 6

Creating DOFObject Instances 6

Making a DOFObject a Provider 7

Implementing the Provider Interface 7

Initiating a Provide Operation 8

Default Provider Behavior 8

Chapter 3: Synchronous Operations 11

Getting Properties Synchronously 11

Provider 11

Requestor 13

Setting Properties Synchronously 15

Provider 15

Requestor 16

Invoking Methods Synchronously 17

Provider 17

Requestor 19

Chapter 4: Exception Handling 21

DOF Exception Types 21

Returning Exceptions in Provider Applications 21

Provider Exceptions 21

Error Exceptions 23

Catching Exceptions in Requestor Applications 24

Chapter 5: Interest and Query Operations 26

Interest Operations 26

Relationship Between Interest and Provide Operations 26
Interest Levels 27

Query Operations **27**

Discovering a Single Provider **27**
Expressing Interest 27
The WaitProvider Query 29

Customized Query Operations **30**
Building a Query 30
Implementing a Query Listener 31

Discovering Multiple Providers **33**
Expressing Interest 33
Initiating a Query Operation 33

Chapter 6: Single-Transaction Asynchronous Operations 35

Getting Properties Asynchronously **35**
Get Operation Listener 36
Calling beginGet 37

Setting Properties Asynchronously **37**
Set Operation Listener 38
Calling beginSet 38

Invoking Methods Asynchronously **39**
Invoke Operation Listener 39
Calling beginInvoke 40

Chapter 7: Ongoing Asynchronous Operations 41

Subscribing to a Property **41**
Providers 41
Subscribe Operation Listener 42
Instantiating DOFSubscription 43

Registering for an Event **44**
Providers 44
Register Operation Listener 45
Instantiating DOFRegistration 46

Chapter 8: Activate-Level Interest 47

Writing the activate Method **47**

Writing the cancelActivate Method **48**

Appendix A: Formatting OIDs 49

GUID Class OIDs **49**

Domain Class OIDs **49**

Email Class OIDs **50**
Formatting OIDs for DOFObjectID Create Methods **50**
Summary **51**

Chapter 1: Introduction

The *Basic DOF Operations Java Programmer's Guide*contains the basic knowledge applications programmers need to get started with the DOF Object Access Library (OAL). The DOF OAL is currently available in Java, C, and C#.

DOF is an object-oriented technology at its core, and it is most specifically modeled on Java language conventions. If you are unfamiliar with object-oriented programming in general or with specific Java definitions of terms such as *field* and *interface*, you may want to familiarize yourself with these before working with any of the OALs, even if you are not working in Java. A basic understanding of Java will help you to better understand DOF.

How to Read This Guide

Although this guide is organized into types of operations, the guide is intended to be instructional and read from cover to cover. Classes and functionality that are used in multiple operations are introduced in the first operation where they occur. This allows them to be introduced in context, making them easier to understand; however, the information is not repeated when they are used a second time. This means that the guide should be used as a tutorial for building knowledge as you read, and is useful as a reference for specific operations only after you have read through it in its entirety.

Before reading this guide, you should be familiar with the content of the Interface Design Concepts Training guide found in the DOF Essentials SDK.

This guide can be used in conjunction with the operations training in the DOF Essentials SDK. The training packages the full source code shown in this guide and may be copied into your own development environment and used for practice. Although samples shown in this guide are based on the training code, they may not align exactly. Functionality required to make the training code compile and run is often omitted from the samples in this guide so that the discussion can focus on the lines of code that are most relevant to performing the DOF operations.

The Time-Based Alarm Interface

To illustrate DOF operations, the code samples in this guide are based on an interface for a time-based alarm system. The following ideas were behind the basic design for this system:

- Many requestors would be somewhat limited devices that include only an alarm (lacking even their own clocks).
- A few requestors would be management applications that can set the alarm time for the other devices.

- Providers would be applications capable of storing the alarm time for multiple devices and sending a notification to those devices when their alarms should be triggered.

To enable you to work with the DOF interface for this system, the system designer who created the interface would give you three types of information:

- A description of the interface items
- The item identifiers for all of the interface items
- Code for the interface

The following sections provide this information.

Description of Interface Items

The interface defines the following five items of functionality:

- **AlarmActive.** This is a writable property that stores a boolean for whether the alarm is set or not. Providers must store the boolean value. Providers must enable requestors to get or subscribe to the value. They must also enable requestors to set the value. A *true* value means the alarm is on.
- **AlarmTimeValue.** This is a read-only property that stores a date-time value. Providers must store the value and enable requestors to get the value. Providers do not need to implement set operations for this value since the property is read-only.
- **SetNewTime.** This is a method that allows requestors to set the alarm time. Requestors will send a new alarm time as an input to the method. Providers must change the AlarmTimeValue to the input value in response to this method. In addition, the method must return the boolean for the active value.
- **AlarmTriggered.** Providers must program their applications to trigger this event when the current system time is equal to the AlarmTimeValue and the AlarmActive property is set to true. Providers must also enable requestors to register to the event. The event has no outputs.
- **BadTimeValue.** Providers must program their applications to throw this exception when a requestor attempts to set an invalid time value in the SetNewTime method.

Item Identifiers

Every item in an interface is assigned an identifier. These item identifiers must be unique within an interface and may be as large as 2^{16}-1; however, because most interfaces have only a handful of items, item identifiers will typically be small. Item identifiers for the Time-Based Alarm interface are shown in the following table.

Item	Identifier
AlarmActive property	1
AlarmTimeValue property	2

Item	Identifier
SetNewTime method	3
AlarmTriggered event	4
BadTimeValue exception	5

Interface Code

In this section, we'll examine the code the system developer created for the interface. Some interface code may be more basic, but our system developer created a number of constants as a convenience for accessing needed information from the interface.

Sample Code

The following is the Java class created for the Time-Based Alarm interface:

```
public class TBAInterface {
   public static final DOFType IS_ACTIVE = DOFBoolean.TYPE;
   public static final DOFType ALARM_TIME = DOFDateTime.TYPE;

   public static final DOFInterface DEF;
   public static final DOFInterfaceID IID = DOFInterfaceID.create
            ("[1:{01000034}]");

   public static final int PROPERTY_ALARM_ACTIVE_ID = 1;
   public static final int PROPERTY_ALARM_TIME_VALUE_ID = 2;
   public static final int METHOD_SET_NEW_TIME_ID = 3;
   public static final int EVENT_ALARM_TRIGGERED_ID = 4;
   public static final int EXCEPTION_BAD_TIME_VALUE_ID = 5;

   public static final DOFInterface.Property PROPERTY_ALARM_ACTIVE;
   public static final DOFInterface.Property PROPERTY_ALARM_TIME_VALUE;
   public static final DOFInterface.Method METHOD_SET_NEW_TIME;
   public static final DOFInterface.Event EVENT_ALARM_TRIGGERED;
   public static final DOFInterface.Exception EXCEPTION_BAD_TIME_VALUE;

   static {
      DEF = new DOFInterface.Builder(IID)
         .addProperty(1, true, true, IS_ACTIVE)
         .addProperty(2, false, true, ALARM_TIME)
         .addMethod(3, new DOFType[] { ALARM_TIME }, new DOFType[] { IS_ACTIVE })
         .addEvent(4, new DOFType[] {})
         .addException(5, new DOFType[] {})
         .build();

      PROPERTY_ALARM_ACTIVE = DEF.getProperty(PROPERTY_ALARM_ACTIVE_ID);
      PROPERTY_ALARM_TIME_VALUE = DEF.getProperty(PROPERTY_ALARM_TIME_VALUE_ID);
      METHOD_SET_NEW_TIME = DEF.getMethod(METHOD_SET_NEW_TIME_ID);
      EVENT_ALARM_TRIGGERED = DEF.getEvent(EVENT_ALARM_TRIGGERED_ID);
      EXCEPTION_BAD_TIME_VALUE = DEF.getException(EXCEPTION_BAD_TIME_VALUE_ID);
   }
}
```

Code Discussion

This class is in the operations training and can be copied and pasted into your project to provide access to the following variables for the DOF interface that you will need in requestor and provider code:

The DOFInterface instance. This is a static class field called DEF, so it can be accessed as TBAInterface.DEF when you need a DOFInterface instance.

The DOFInterfaceID instance. This is a static class field called IID, so it can be accessed as TBAInterface.IID when you need a DOFInterfaceID instance. A DOFInterfaceID instance can also always be obtained from the DOFInterface instance using the *DOFInterface.getInterfaceID()* method. . Samples in this guide do not show this.

Interface items. As shown, the TBAInterface class defines constants for all the items in the interface, which will be used in sample code in this guide. If we did not have access to these constants, we could use the getter methods shown in the class to get the items.

Chapter 2: Basic Setup

Before you can begin writing operations, you must set up the basic code that makes an application capable of being either a provider or a requestor. Both providers and requestors need an instance of DOFObject to initiate or process operations involved with interface functionality. Instantiating the DOFObject class requires that you first have an instance of two other classes: DOFSystem and DOFObjectID. This is covered in later sections.

In preparation for the rest of the guide if you want to use the operations training code to set up a working environment for experimentation, do the following:

1. Import the following classes from the operations training or copy them into your package (these are located in the *shared* directory):
   - DOFTrainingDOFAbstraction.java
   - TBAInterface.java
2. Instantiate DOFTrainingDOFAbstraction in an initialization routine using the default constructor.
3. Call *DOFTrainingDOFAbstraction.destroy* in a shutdown routine.

Obtaining DOFSystem Instances

To create a DOFObject, you must use a method of DOFSystem, so you need an instance of DOFSystem on which to call the method. In general, creating DOFSystem instances is a concern for system developers. This guide doesn't cover the creation of DOFSystem instances, but assumes that a system developer will pass it to you or will have created a class with a method in it for creating the system.

If you are using the operations training code in conjunction with reading this guide, it contains a DOF abstraction class that has a method in it for creating a system, and which you can use in your project to get a DOFSystem instance.

If you copy DOFTrainingDOFAbstraction.java into your project and instantiate this class in an initialization routine, you can use the *DOFSystem createSystem(String)* method to get an instance of DOFSystem. The input parameter represents a name for the system and can be anything you want; however, if you create multiple systems, you should use unique names.

Sample code in this guide assumes the DOFSystem instance has already been obtained and simply refers to instances by the variable name *system*.

Note: Both a DOF and a DOFSystem are actually required to create a DOFObject. This guide assumes you are using the DOF abstraction code in the training, which creates a DOF for you. For more information about DOF or DOFSystem, refer to the *DOF Connectivity Programmer's Guide* in the DOF Essentials SDK.

Creating DOFObjectID Instances

The DOFObjectID class in the DOF libraries is used to create unique object identifiers (OIDs) for any type of object that needs one. In the simplest scenario with a single provider and a single requestor, the OID identifies the provider:

- The provider creates a DOFObjectID using an OID that identifies itself as an DOF object in the DOF Object Model. Its DOFObjectID instance essentially states, "I am this object."
- The requestor uses the *same* OID to create its DOFObjectID. Its DOFObjectID instance essentially states, "This is the provider I am going to want to request functionality from."

In more advanced scenarios, each requestor might need its own OID, as well. For example, many services are essentially provider applications capable of creating multiple DOFObjects that provide the same functionality for multiple requestors, while maintaining separate state or database information for each individual requestor.

In a production environment, OIDs should be carefully managed and assigned to objects, and the individual in charge of managing the OIDs may tell you what to use; however, if you are experimenting in an attempt to learn DOF, you may need to create your own. *Appendix A: Formatting OIDs* contains directions on how you can create and properly format an essentially unlimited number of unique OIDs if you own your own registered Internet domain, or how you can use GUIDs to create OIDs.

Sample Code

The DOFObjectID class contains static *create* method overloads for instantiating a DOFObjectID. The following example shows how to instantiate a DOFObjectID using the *DOFObjectID.create(String)* method.

```
DOFObjectID providerID = DOFObjectID.create("[3:provider@opendof.org]");
```

Code Discussion

The string passed to this method must be in the standard string format prescribed for OIDs, which is explained in *Appendix A: Formatting OIDs*.

Creating DOFObject Instances

The DOFObject class essentially represents a provider on the network. Even in requestor code, it is conceptualized as the provider you want to talk to. In the DOF Object Model, the DOFObject instance is the DOF object. The methods that enable you to initiate and process operations are contained in this class. In addition, the DOFObject is used by the internal library for addressing and routing. The create methods for DOFObject require an instance of DOFObjectID.

Sample Code

The following example shows how to instantiate DOFObject:

```
DOFObject provider = system.createObject(providerID);
```

You must destroy the DOFObject when you are finished with it. The following shows how to destroy DOFObject instances:

```
provider.destroy();
```

Code Discussion

While you must destroy DOFObject, the Java OAL contains a master *DOF.destroy* method that will cancel all operations and destroy all objects that have been created on a node. (For example, it also destroys the DOFSystem.) For resource management purposes, you may want to create and destroy DOFObject instances throughout the life of the program, but if the application is likely to use the DOFObject for as long as it is running, calling *DOF.destroy* in your shutdown routine is adequate to cancel all operations and destroy all objects your application initiates or creates.

If you are using the DOFTrainingDOFAbstraction class from the operations training, calling *destroy* on your instance calls *DOF.destroy*.

If you are writing a requestor application, you are ready to begin writing methods that initiate operations once you have a DOFObject instance. However, if you are writing a provider application, a little more preparation is required, as described in the next section.

Making a DOFObject a Provider

For a DOFObject to act as a provider, you must implement an interface that enables the DOFObject to listen for and respond to operations from requestors. This section describes only how to set up the interface implementation. Later sections of the guide describe how to implement the callbacks in the interface.

Implementing the Provider Interface

The Java OAL includes a DOFObject.Provider interface that you may implement fully when customized behavior is needed for all the callbacks in the Provider interface. However, it also contains a DOFObject.DefaultProvider class that implements the Provider interface and provides default functionality for operations you do not need to support. This can enable you to create more elegant code and prevents you from having to implement methods that you do not need. For example, if you are providing an DOF interface that does not contain any methods, then using DefaultProvider prevents you from having to implement your own *invoke* methods.

Sample Code

Preparing the provider is a simple matter of creating a class that extends the DOFObject.DefaultProvider class, although where needed, you can instead implement DOFObject.Provider:

```
public class ProviderListener extends DOFObject.DefaultProvider {
}
```

Initiating a Provide Operation

Chapter 8: Activate-Level Interest will cover how to begin providing an interface in response to a demand for it; however, for functionality discussed in earlier chapters, you should program your provider to begin providing when it starts up.

The method needed to begin providing is *DOFObject.beginProvide (DOFInterface, int, DOFObject.Provider, java.lang.Object).*

Sample Code

The following is a sample of the *beginProvide* method:

```
DOFOperation.Provide provideOperation = provider.beginProvide (TBAInterface.DEF,
         DOF.TIMEOUT_NEVER, new ProviderListener(), null);
```

The operation is canceled when you destroy the DOFObject, the DOFSystem, or the DOF, but it some cases it may be necessary to explicitly call the *cancel* method:

```
provideOperation.cancel();
```

Code Discussion

The following describes the parameters of the *beginProvide* method:

- **DOFObject.** The first parameter is the DOFObject instance created earlier. Because the C OAL has an object-oriented model, but C is not an object-oriented language, the first parameter of many functions in the C OAL is an instance of the "class" the function belongs to.
- **DOFInterface.** This parameter is the DOFInterface instance that was described in the section on *The Time-Based Alarm Interface*.
- **int.** This parameter represents a "timeout." In the case of an asynchronous operation like provide, the timeout is how long the operation will remain active without completing and shutting down. We want our provider to continue providing for as long as it is on, so we used the "timeout never" constant provided by the library.
- **DOFObject.Provider.** This argument must be an instance of a class that implements DOFObject.Provider.
- **Object.** This parameter offers additional flexibility in the library. You can think of it as an "attachment" that your provider application would receive every time the callbacks in the provider interface are called. So if you wanted to receive custom information in all of the provider interface callbacks, you would create a custom class and pass an instance of that class in this argument. This is an advanced and highly customized use of the library, so the rest of this guide ignores "context" parameters. In our samples, we simply pass null arguments.

Default Provider Behavior

Although this guide eventually covers how to implement functionality in each of the methods or functions in the provider interface, it is useful to be aware of the default behavior that occurs if you extend the DefaultProvider.

Get, Set, Invoke, and Session Calls

When the library receives *get*, *set*, and *invoke* operations, it first checks these operations against the DOF interface definition. If the interface does not contain a corresponding property or method, the provider application's method is never called, according to the following specifications:

- Get is never called if the DOF interface defines no readable properties.
- Set is never called if the DOF interface defines no writable properties.
- Invoke is never called if the DOF interface defines no methods.

If the interface does define properties or methods that correspond with these operations, and you do not override the get, set, or invoke methods, the OAL returns a "not supported" exception. The provider, however, is in breach of its contract. A provider application should always override the default behavior of the *get*, *set*, or *invoke* callbacks if the DOF interface defines corresponding items of functionality.

The default behavior for the session method is to return a "not supported" exception. Providers should implement session when they provide an interface that is defined as a base interface for sessions. Sessions are not further discussed in this guide.

Subscribe and Register Calls

As with the get, set, and invoke operations, the subscribe and register functions are not called under the following conditions:

- Subscribe is never called if the DOF interface defines no readable properties.
- Register is never called if the DOF interface defines no events.

However, the default behavior in these methods is to automatically accept the subscribe or register operation. So, unlike with *get*, *set*, and *invoke*, the provider does not need to implement additional functionality in the *subscribe* and *register* callbacks to support subscribe and register operations. However, if you are implementing DOFObject.Provider rather than extending DOFObject.DefaultProvider, the *subscribe* and *register* methods must contain, at minimum, the following call to prevent the requestor from timing out:

```
request.respond();
```

Subscribe, Register, and Session Complete Calls

These callbacks are useful places to clean up any resources allocated in association with subscribe, register, or session operations. The default behavior in these callbacks is to do nothing, but you should override the default behavior if you need to perform cleanup associated with the operations.

Complete Call

This method is called when a provide operation completes, and it can be used to clean up resources. The default behavior for this callback is to do nothing, but you should override the default behavior if you need to clean up resources associated with the provide operation.

In addition, if the *subscribeComplete*, *registerComplete*, or *sessionComplete* callbacks contain functionality associated with cleaning up resources, it is a good practice to also clean up those resources in the *complete* callback for the provide operation. The library cannot call *subscribeComplete*, *registerComplete*, or *sessionComplete* after *complete* has been called for the provide operation, and it is possible for these calls to be delivered out of order.

Chapter 3: Synchronous Operations

The OAL provides synchronous operations for getting or setting an DOF interface property value and for invoking an DOF interface method. In this chapter, you will learn how to implement these three types of operations for both providers and requestors.

Getting Properties Synchronously

A requestor initiates a synchronous get operation using the *get* method of the DOFObject class. To properly return a value to the requestor, the provider must override the *get* callback of the DOFObject.Provider interface. The sequence of a synchronous get operation is shown in the following diagram.

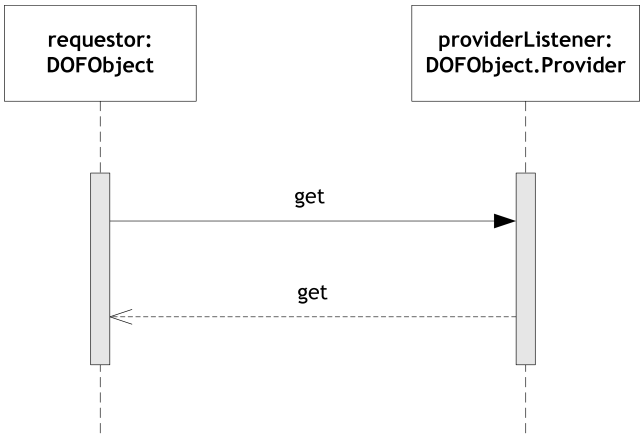

Figure 1. Synchronous Get Operation Sequence Diagram

Provider

For a provider to handle any interface property, it must obviously store the property's value, usually in a class field. The code samples shown in this guide use the two properties of the Time-Based Alarm interface described in the introduction. The type definition for the AlarmActive property is a boolean, so you should assume when looking at the samples that earlier code has initialized a boolean called *isActive* and set its initial value to false. For the AlarmTime value, assume a Date was initialized with the variable name *alarmTime* and set to zero.

With the property initialized, you can now create the functionality that will enable the provider to return the value of that property to requestors that ask for it.

Sample Code

The following sample shows how to override the *DOFObject.Provider.get* method to return the properties in the Time-Based Alarm interface:

```
@Override
public void get(Provide operation, DOFRequest.Get request, Property property) {
```

```
    if(property.equals(TBAInterface.PROPERTY_ALARM_ACTIVE)){
       DOFValue dofBoolean = new DOFBoolean(isActive);
       request.respond(dofBoolean);
    } else if (property.equals(TBAInterface.PROPERTY_ALARM_TIME_VALUE)){
       DOFValue dofDateTime = new DOFDateTime(alarmTime);
       request.respond(dofDateTime);
    }
}
```

Code Discussion

The following steps provide directions for coding a provider's *get* callback, using the sample above for an example:

1. Compare the interface item passed to you with the property defined in the DOF interface.

   Obviously, an interface definition can contain more than one of any item type, so you should compare what was passed to you to each of the interface properties to determine which value to return. In addition, it is a good practice to make this check even when you have only one item of a type defined in an interface.

2. Create a DOFValue instance from the stored property value.

   A DOFValue instance is what the library needs to return to the requestor, so the first step is to create a DOFValue instance from a stored primitive type. Primitive types have corresponding classes in the Java OAL that implement the DOFValue interface. So to create a DOFValue instance from a boolean primitive, for example, you use a constructor for the DOFBoolean class.

   Some types defined in DOF interfaces, such as strings and blobs, may require additional parameters to construct, such as encoding and maximum length for strings. Each DOF interface defines the valid values or ranges for these types.

3. Use a DOFRequest method to return the DOFValue instance to the requestor.

   The library passes a DOFRequest object to the provider in the *get* callback. This is actually an intance of a DOFRequest nested class that corresponds with the operation type—DOFObject.Get. To return the property value to the requestor, call the *DOFRequest.Get.respond(DOFValue)* method and pass the DOFValue created in step 1 as its argument.

   Whenever you override the provider's *get* method, you *must* call *respond* to return either a DOFValue or an exception. If you do not, the requestor's operation will time out.

Requestor

To get the value of a property, requestors need to use the *DOFObject.get* method. The Java OAL includes two overloads:

- *DOFObject.get(DOFInterface.Property, int)*
- *DOFObject.get(DOFInterface.Property, DOFOperation.Control, int)*

Using an operation control is an advanced topic not discussed in this basic programming guide. Samples shown in this guide use the method overloads that do not take DOFOperation.Control arguments.

The next chapter discusses how to add exception handling. In this chapter, the samples throw the exception.

Sample Code

The following sample shows how to use the *DOFObject.get* method to get the value of the AlarmActive property:

```
public Boolean sendGetRequest() throws DOFException{
   DOFInterface.Property property = TBAInterface.PROPERTY_ALARM_ACTIVE;

   DOFResult<DOFValue> result;
   DOFValue value;
   DOFBoolean dofBoolean;
   boolean unwrappedResult;

   result = provider.get (property, 10000);
   value = result.get();
   dofBoolean = (DOFBoolean)value;
   unwrappedResult = dofBoolean.get();

   return unwrappedResult;
}
```

Code Discussion

The following steps provide directions for coding your own requestor's *get* method, using the samples above for an example:

1. Initialize the desired interface property.

2. Call the requestor's *get* method.

   This returns a DOFResult, so use it to initialize an instance of DOFResult. Pass the property created in Step 1 to this method.

The int parameter represents a timeout, which is a time in milliseconds that the application will wait to receive the result of the operation from the property. Because this is a blocking call, you probably do not want to set the timeout too high, but it should be long enough for the requestor to communicate with the provider across the network. The samples have set the timeout to 10 seconds, but you may need to adjust this to a higher time if your network communication is slow. You would also want to use a higher time in cases where the property value is a large chunk of data, such as a large blob.

3. Extract a DOFValue from the returned DOFResult.

   The DOFResult class has a method that returns the DOFValue. This is *DOFResult.get()*.

4. Cast the DOFValue to a class that represents a primitive type.

   As described in the provider section, DOFValue is an interface in the Java OAL, implemented by multiple classes that represent various primitive types. The classes do not represent only the primitives available in Java, they represent all the types that can be defined in an DOF interface. For example, while Java does not have unsigned primitives, DOF interfaces do. To know which class to use in this step, you must rely on the description of the DOF interface items that was given to you by the interface designer. Our description of the Time-Based Alarm interface stated that the AlarmActive property was a boolean type, so in the samples, we cast DOFValue to a DOFBoolean.

5. Unwrap the DOFValue.

   Each of the classes that implement DOFValue and correspond to primitive types contains a *get()* method that returns a primitive type. Thus, in our sample *DOFBoolean.get* returns a Java boolean primitive. However, because Java does not support unsigned primitives, you will want to be aware of the return types for the *get* methods. For example, while *DOFInt8.get* returns a Java byte, *DOFUInt8.get* returns a Java short. Refer to the API reference documentation for information on return types.

Setting Properties Synchronously

A requestor initiates a synchronous set operation using the *set* method of the DOFObject class. To properly return a value to the requestor, the provider must override the *set* method in the Provider interface. The sequence of a synchronous set operation is shown in the following diagram.

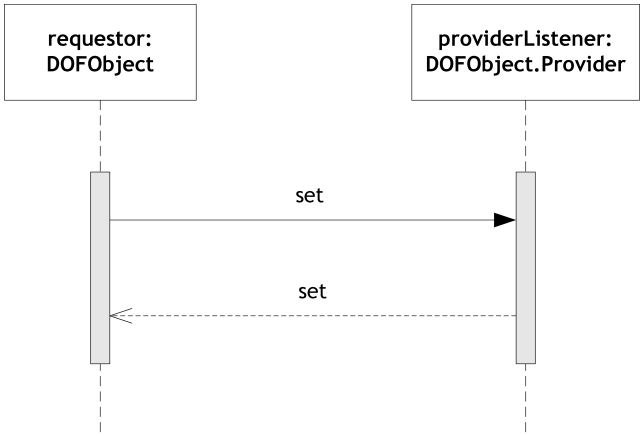

Figure 2. Synchronous Set Operation Sequence Diagram

Provider

The provider often needs to implement functionality in response to a set operation, such as changing the physical state of a device or triggering user notifications. Such application logic is specific to the device. For example, our sample code shows how to set the AlarmActive property in the Time-Based Alarm interface. This property determines whether or not the AlarmTriggered event will be triggered at the time stored in the AlarmTimeValue property. Application logic requires changing the stored value of the property in response to a set operation.

Outside of application logic, DOF requires only that the provider respond to the requestor. Very few calls in the following samples should be new to you.

Sample Code

The following sample shows how to override the *DOFObject.Provider.set* method to return the value of the AlarmActive property:

```
@Override
public void set(Provide operation, DOFRequest.Set request, Property property,
          DOFValue value) {
   if(property.equals(TBAInterface.PROPERTY_ALARM_ACTIVE)){
      DOFBoolean dofBoolean = (DOFBoolean)value;
      isActive = dofBoolean.get();
      request.respond();
   }
}
```

Code Discussion

The following steps provide directions for coding a provider's *set* method, using the samples above for an example:

1. Cast the DOFValue that was passed to you to a class that represents a primitive type.
2. Unwrap the DOFValue and set the stored property value equal to the unwrapped value.
3. Use a DOFRequest method to respond to the requestor.

   Although no values need to be returned, you must call *DOFRequest.Set.respond* to send an acknowledgement to the requestor and prevent its operation from timing out.

Requestor

To set the value of a property, requestors need to call *DOFObject.set (DOFInterface.Property, DOFValue, int)*.

Sample Code

The following sample shows how to use the *DOFObject.set* method to set the value of the AlarmActive property:

```
public void sendSetRequest(boolean isActive) throws DOFException{
   DOFInterface.Property property = TBAInterface.PROPERTY_ALARM_ACTIVE;
   DOFBoolean setValue;

   setValue = new DOFBoolean(isActive);
   provider.set(property, setValue, 10000);
}
```

Code Discussion

The following steps provide directions for coding a requestor's *set* method, using the samples above for an example:

1. Convert the primitive input type to a DOFValue.
2. Call the requestor's *set* method and pass it the new value to be set.

   As with the get operation, the int parameter is a timeout that represents how long you want the application to wait to complete.

   Now that we have seen code for both get and set operations, it is a good time to discuss the DOFResult class. All of a requestor's methods for initiating synchronous operations return a DOFResult; however, each DOFResult contains slightly different information.

Java-language generics are used to represent this difference. The *get* method returns a DOFResult<DOFValue>; the *set* method returns DOFResult<Void>; and the *invoke* method that is covered in the next section returns DOFResult<List<DOFValue>>. As you will recall, when the requestor performed its get operation, it used the *DOFObject.get* method to initialize DOFResult and then used the *DOFResult.get* method to retrieve the DOFValue contained in the result. In our *set* method sample, the returned DOFResult is ignored, because we know it doesn't contain a DOFValue; however, we could initialize it if we needed to perform some other functionality, such as using the *DOFResult.getProviderInfo* method to retrieve information about the provider.

Invoking Methods Synchronously

A requestor initiates a synchronous invoke operation using the *invoke* method of the DOFObject class. To properly implement an invoke operation, the provider must use the *invoke* method in the Provider interface to execute the functionality defined for the method in the DOF interface and return any required values to the requestor. The sequence of a synchronous invoke operation is shown in the following diagram.

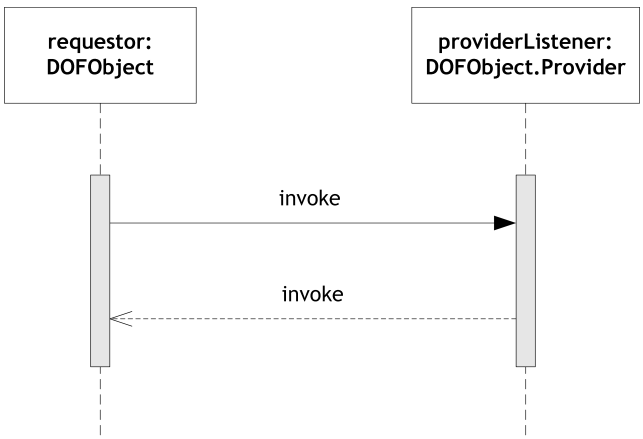

Figure 3. Synchronous Invoke Operation Sequence Diagram

Provider

Because DOF interface methods are methods in the true programming sense, implementing the functionality they define can be as complex or simple as implementing any programming method. Clearly, the SetNewTime method in our example Time-Based Alarm interface is a reasonably simple one: set the AlarmTime property and return the value of the AlarmActive property.

You may wonder why an interface designer would make a method like this one rather than just making the AlarmTime property writable and allowing the requestor to set it. However, this is actually a fairly common type of method in DOF interfaces, because it allows the separation of permissions in the DOF security model. Many requestors could be granted write permission for the interface, and they could then set the AlarmActive property, but they would still be unable to change the read-only AlarmActive property unless they also had execute permission, allowing them to invoke the method.

For a provider to handle any interface property, it must obviously store the property's value, usually in a class field. The code samples shown in this use the AlarmActive property of the Time-Based Alarm interface described in the introduction. The type definition for the AlarmActive property is a boolean, so you should assume when looking at the samples that earlier code has initialized a boolean called *isActive* and set its initial value to false.

The provider needs to store the value of the second interface property in a class field. You should assume when looking at the samples that earlier code has initialized a DateTime called *alarmTime* and set its initial value to null.

Sample Code

The following sample shows how to override the *DOFObject.Provider.invoke* method to perform the functionality defined for the SetNewTime method:

```
@Override
public void invoke(Provide operation, DOFRequest.Invoke request, Method method,
         List<DOFValue> parameters) {
   if(method.equals(TBAInterface.METHOD_SET_NEW_TIME)){
      DOFValue inputParameter;
      DOFDateTime dofDateTime;
      Date unwrappedInputParam;
      DOFValue returnValue;

      /* Handling Method Input */
      inputParameter = parameters.get(0);
      dofDateTime = (DOFDateTime)inputParameter;
      unwrappedInputParam = dofDateTime.get();
      alarmTime = unwrappedInputParam;

      /* Sending Method Output */
      returnValue = new DOFBoolean(isActive);
      request.respond(returnValue);
   }
}
```

Code Discussion

The following steps provide directions for coding a provider's *invoke* method, using the samples above for an example:

1. Get the DOFValue instances that represent the input parameters out of the List.

   When methods have no input parameters, you can skip steps 1 through 3.

Because a method can have multiple input parameters, input parameters are always passed to the invoke method as a List, even if the method has zero or one input parameters.

Because our sample method has only one input parameter, we know it is at offset zero and so we retrieve it and use it to initialize a DOFValue instance. When DOF interface methods have multiple input parameters, they must be ordered as the interface defines.

2. Cast the DOFValue to a class that represents primitive types.
3. Unwrap the DOFValue.
4. Use input parameters as defined by the DOF interface.
   For our sample interface, this step is as simple as setting the alarmTime variable equal to the unwrapped input parameter, although functionality may be more complex for many interface methods.

5. Create DOFValue instances for the output parameters.
   All output parameters must be DOFValue instances. Because our sample method has only one output parameter, the sample code simply creates a single DOFBoolean.

   When methods have no output parameters, you can skip steps 5 and 6.

6. Add the DOFValue instances to a List.

   This step is optional and not shown in the sample above, because the library contains a method that takes varargs, which was used to return the single output parameter. If you have multiple output parameters, you must provide them in the order defined by the DOF interface.

7. Use a DOFRequest method to respond to the requestor.

   Use *DOFRequest.Invoke.respond (DOFValue... values)* or *DOFRequest.Invoke.respond(List<DOFValue> values)*.

   Even when the method has no output values, you *must* call *DOFRequest.Invoke.respond* to prevent the requestor from timing out.

Requestor

To invoke a method, requestors need to call *DOFObject.invoke*. The Java OAL includes two overloads (in addition to two that take a DOFOperation.Control, which this guide does not cover):

- *DOFObject.invoke(DOFInterface.Method, int, DOFValue...)*
- *DOFObject.invoke(DOFInterface.Method, List<DOFValue>, int)*

As in similar methods, the int parameter represents the timeout.

Sample Code

The following sample shows how to use the *DOFObject.invoke* method to invoke the SetNewTime method:

```
public Boolean sendInvokeRequest(Date alarmTime) throws DOFException{
   DOFInterface.Method method = TBAInterface.METHOD_SET_NEW_TIME;
   DOFValue inputParameter;
   DOFResult<List<DOFValue>> result;
   List<DOFValue> returnValues;
   DOFValue returnValue;
   DOFBoolean dofBoolean;
   boolean unwrappedReturnVal;

   /* Sending Method Input */
   inputParameter = new DOFDateTime(alarmTime);
   result = provider.invoke (method, 10000, inputParameter);

   /* Handling Method Output */
   returnValues = result.get();
   returnValue = returnValues.get(0);
   dofBoolean = (DOFBoolean)returnValue;
   unwrappedReturnVal = dofBoolean.get();

   return unwrappedReturnVal;
}
```

Code Discussion

The following steps provide directions for coding a requestor's *invoke* method, using the samples above for an example:

1. Get the method from the interface.
2. Convert the input parameter to a DOFValue instance.

   When methods have no input parameters, you can skip steps 2 and 3.

3. Add the DOFValue instance to a List.

   This step is optional and not shown in the sample above, because you can use the *invoke* method that takes varargs instead of the one that takes a List<DOFValue>. If you have multiple input parameters, you must provide them in the order defined by the DOF interface.

4. Call the requestor's *invoke* method.

   When methods have no output parameters, you can skip steps 5 through 8.

5. Get the List of output parameters from the DOFResult.
6. Get the DOFValue instance that represents the output parameter from the List.
7. Cast the DOFValue to a class that represents primitive types.
8. Unwrap the DOFValue.

Chapter 4: Exception Handling

For simplicity, the operations shown in the previous section did not do exception handling. In a production environment, you should never perform these operations without proper exception handling. This section covers the following topics:

- DOF exception types
- Modifying provider code to throw library exceptions
- Modifying requestor code to catch library exceptions

DOF Exception Types

The DOF OALs define two basic types of exception:

- **DOFProviderException.** This class represents exceptions defined by an DOF interface.
- **DOFErrorException.** This class represents enumerated exceptions defined in the DOF library.

Returning Exceptions in Provider Applications

This section provides directions for creating provider and error exceptions and returning them to a requestor.

Provider Exceptions

The sample code in this section shows the provider's *invoke* method shown in the previous chapter. It has been modified to check for the condition that should cause a provider exception to be thrown and to throw the exception when that condition occurs. The discussions after the samples cover only the new information.

To create provider exceptions, you can use the following constructors:

- *DOFProviderException(DOFInterface.Exception, DOFValue...)*
- *DOFProviderException(DOFInterface.Exception, List<DOFValue>)*

Sample Code

The following sample shows how to add a provider exception to the provider's *invoke* method:

```
@Override
public void invoke(Provide operation, DOFRequest.Invoke request, Method method,
         List<DOFValue> parameters) {
   if(method.equals(TBAInterface.METHOD_SET_NEW_TIME)){
      DOFValue inputParameter;
      DOFDateTime dofDateTime;
      Date unwrappedInputParam;

      /* Handling Method Input */
      inputParameter = parameters.get(0);
```

```
        dofDateTime = (DOFDateTime)inputParameter;
        unwrappedInputParam = dofDateTime.get();

        if(unwrappedInputParam.before(new Date())){
           DOFInterface.Exception interfaceException;
           DOFProviderException ex;

           interfaceException = TBAInterface.EXCEPTION_BAD_TIME_VALUE;
           ex = new DOFProviderException(interfaceException);
           request.respond(ex);
        }else{
           DOFValue returnValue;

           alarmTime = unwrappedInputParam;

           /* Sending Method Output */
           returnValue = new DOFBoolean(isActive);
           request.respond(returnValue);
        }
     }
}
```

Code Discussion

The following steps provide directions for creating a provider exception, using the samples above for an example:

1. Check for the error condition.

   In the Time-Based Alarm interface, the exception should be thrown if the time provided by the requestor is earlier than the current time. The code uses an if statement to check for this condition.

2. Initialize the exception.
3. Create DOFValue instances for the output parameters.

   When exceptions have no output parameters, you can skip steps 3 and 4. The samples do not show these steps, because the BadTimeValue exception defined in the Time-Based Alarm interface has no output parameters.

4. Add the DOFValue instances to a List.

   This is optional, because you can use the function that accepts varargs instead of the one that accepts a List, if you choose. Output parameters should be provided in the order defined by the DOF interface.

5. Create a DOFProviderException.

   Use the constructors shown in the introduction to this section. When the exception has no output parameters, you can use the constructor that accepts varargs by not passing any arguments. You could also create an empty list and return it.

6. Use a DOFRequest method to respond to the requestor.

The Java OAL has a *respond* method overload that takes a DOFProviderException.

Error Exceptions

The code samples in this section show the same get operations shown earlier in this chapter, but they have been modified to throw an exception as a default case. Because this code should never be reached, you could also use an assert in this situation, but the exception is also a logical default case. The directions after the samples do not walk through the entire code sample again, they simply cover the new information.

The Java OAL provides several constructors for DOFErrorException, but the usage of these should be fairly straightforward. The samples use the *DOFErrorException(int, String)* constructor.

Sample Code

The following sample shows how to add an error exception to the provider's *get* method:

```
@Override
public void get(Provide operation, DOFRequest.Get request, Property property) {
   if(property.equals(TBAInterface.PROPERTY_ALARM_ACTIVE)){
      DOFValue dofBoolean = new DOFBoolean(isActive);
      request.respond(dofBoolean);
   } else if (property.equals(TBAInterface.PROPERTY_ALARM_TIME_VALUE)){
      DOFDateTime dofDateTime = new DOFDateTime(alarmTime);
      request.respond(dofDateTime);
   } else {
      DOFErrorException ex;

      ex = new DOFErrorException (DOFErrorException.NOT_SUPPORTED, "This provider
               does not support the requested action for the specified interface
               item.")
      request.respond(ex);
   }
}
```

Code Discussion

The following steps provide directions for creating an error exception, using the samples above for an example:

1. Create a DOFErrorException.

   Use the constructors shown in the introduction to this section. The int parameter must be filled by an error code argument. The error codes are defined as DOFErrorException class fields.

2. Use a DOFRequest method to respond to the requestor.

   The Java OAL has a *respond* method overload that takes a DOFErrorException.

Catching Exceptions in Requestor Applications

All of the methods in the DOF OALs that initiate operations are capable of throwing exceptions, so requestors should be prepared to catch exceptions whenever they initiate operations. In addition, requestors need to be aware of the provider exceptions defined in DOF interfaces and the conditions that trigger them, so they can handle them correctly.

The exceptions defined by the OALs obviously do not cover all error conditions that can occur in code. In general, the exceptions defined in the OALs are those that can occur in communication between requestors and providers. Programmers should handle general programming errors according to best practices.

The Java OAL divides exceptions into the following class hierarchy:

DOFException extends java.lang.Exception
    DOFProviderException extends DOFException
    DOFErrorException extends DOFException
        DOFSecurityException extends DOFErrorException

Exception handling in the Java OAL follows common Java-language practice, and should be very straightforward for Java programmers. The DOFSecurityException class is not covered in this guide. Refer to the *Basic DOF Security Programmer's Guide* in the DOF Essentials SDK for more information on security exceptions.

Sample Code

The following sample shows how to add exception handling to the requestor's *invoke* method that was shown in the previous chapter.

```
public Boolean sendInvokeRequest(){
   DOFInterface.Method method;
   DOFValue inputParameter;
   DOFResult<List<DOFValue>> results;
   List<DOFValue> returnValues;
   DOFValue returnValue;
   DOFBoolean dofBoolean;
   boolean unwrappedReturnVal;

   try{
      /* Sending Method Input */
      method = TBAInterface.METHOD_SET_NEW_TIME;
      inputParameter = new DOFDateTime(alarmTime);
      results = provider.invoke (method, 10000, inputParameter);

      /* Handling Method Output */
      returnValues = results.get();
      returnValue = returnValues.get(0);
      dofBoolean = (DOFBoolean)returnValue;
      unwrappedReturnVal = dofBoolean.get();

      return unwrappedReturnVal;

   } catch(DOFProviderException ex){
```

```
            DOFInterface.Exception exception = ex.getInterfaceException();
            if(exception.equals(TBAInterface.EXCEPTION_BAD_TIME_VALUE)){
               System.out.println("Received BadTimeValue exception.");
            }else{
               System.out.println("Received unknown provider exception.”);
            }
            return null;

        } catch (DOFErrorException ex) {
            int errorCode = ex.getErrorCode();
            System.out.println("Exception: " + ex.getErrorCodeString(errorCode) + " " +
                     ex.getMessage());
            return null;

        } catch (DOFException ex) {
            int errorCode = ex.getErrorCode();
            System.out.println("Exception: " + ex.getErrorCodeString(errorCode) + " " +
                     ex.getMessage());
            return null;
        }
    }
```

Code Discussion

The following steps outline the changes that are shown in the previous sample:

1. Surround methods that initiate operations in try-catch blocks.
2. Catch error exceptions.

Chapter 5: Interest and Query Operations

Before moving on to the asynchronous operations described in the next two chapters, it is necessary to understand how to set up interest and query operations. You will need interest and query operations if you want to program your requestor to work with multiple providers in the next chapter. And performing an interest operation is required before you can perform the operations described in *Chapter 7: Ongoing Asynchronous Operations*.

Interest Operations

Requestors may need to perform interest operations for a number of reasons. The following use cases are discussed in this guide:

- To discover whether a specific provider is currently running and connected to the network
- To discover multiple providers of an interface
- To support operations that require interest
- To cause a provider that is capable of providing an interface on demand to begin providing it

The following are additional reasons to perform interest operations, although these are beyond the scope of the application developer and are not covered in this guide:

- To cause datagram connections between a requestor and provider to be upgraded to streaming connections
- To optimize network traffic by mapping a route between a requestor and provider, although this must be carefully balanced against the fact that interest operations flood the network and are stored in state on every network node. Use of interest operations where not otherwise required by an operation should be carefully managed by a system designer.

Relationship Between Interest and Provide Operations

Interest is a requestor operation. The provider operation that corresponds with it is a provide operation. The following are true of interest and provide operations:

- A requestor sends an interest operation. Interest operations always flood the network and are stored in state on every node until the interest operation times out or is canceled.
- When a new node connects to an DOF network, the node it connects to automatically forwards its interest table so that new nodes also have all interest operations stored in state.
- Interest operations are sent again every hour (by default) until they are canceled or time out.

- A provider stores its provide operation in its local state, but sends it across the network only when it receives a matching interest operation. Provide operations are routed directly to requestors that have sent corresponding interest operations.

Interest Levels

A requestor can express the following enumerated interest levels:

- **WATCH.** This level of interest enables requestors to discover providers that are already providing an interface. It also causes direct routes to be set up between providers and interested requestors.
- **ACTIVATE.** This level of interest includes all the functionality of the WATCH level. In addition, it causes providers that are capable of providing an interface but not already providing it to begin providing it.
- **CONNECT.** This level of interest includes all the functionality of the ACTIVATE level. In addition, it causes streaming connections to be established between all nodes between the requestor and the provider, if those nodes are capable of establishing streaming connections.

This chapter covers only watch-level interest. Activate-level interest is covered in a later chapter. Connect-level interest is not covered in this guide.

Query Operations

A requestor uses a query operation to parse the provides it receives in response to interest. It is important to remember that a query operation is a *local* operation. It is not sent to other nodes. It will not search the network for providers. It finds only provide operations that the requestor is storing in state on its local system. Because the system will not store provide operations without a matching interest operation, queries are useful only in conjunction with interest.

Discovering a Single Provider

One purpose of using watch-level interest to discover a specific provider is to ensure that the provider is available before you begin sending other operations across the network. The simplest way to do this is to send an interest operation, and then call *DOFSystem.waitProvider*. This method is a query operation shortcut. It returns a DOFObject representing a provider that matches the interest operation.

Expressing Interest

The full signature for the method used to express interest is *DOFSystem.beginInterest (DOFObjectID, DOFInterfaceID, DOFInterestLevel, int, DOFSystem.InterestOperationListener, Object).*

Sample Code

The following sample shows how to express watch-level interest:

```
DOFOperation.Interest interestOp;
```

```
...

interestOp = system.beginInterest(providerID, TBAInterface.IID,
         DOFInterestLevel.WATCH, DOF.TIMEOUT_NEVER, null, null);

...

interestOp.cancel();
```

Code Discussion

The following steps explain the sample code above:

1. Declare a DOFOperation.

   Specifically, declare a DOFOperation.Interest. The declared operation needs to be used in multiple blocks, so it should be a class field.

2. Call the *beginInterest* method.

   Although the sample does not show this, the begin interest call would likely be contained within a block, such as an initialization routine or other method. The following describes the parameters used in the sample:

   - **DOFSystem.** The method is called on the instance of DOFSystem obtained according to the instructions described in *Chapter 2: Basic Setup*.
   - **DOFObjectID.** Pass a DOFObjectID created according to the instructions in *Chapter 2: Basic Setup*. You can use the library's DOFObjectID.BROADCAST constant to express interest in any provider of the specified interface. Sample code using this constant is shown later in this chapter under *Discovering Multiple Providers*.
   - **DOFInterfaceID.** Pass a DOFInterfaceID instance. The one we passed was described in *The Time-Based Alarm Interface*. You can use the library's DOFInterfaceID.WILDCARD constant to express interest in all interfaces provided by the provider with the specified DOFObjectID.

     Note: Using both the broadcast DOFObjectID and wildcard DOFInterfaceID creates a broad interest in any provider of any interface.

   - **DOFInterestLevel.** Interest levels are enumerated. To initiate watch-level interest, the argument should be passed exactly as shown.
   - **int.** For an interest operation, the timeout parameter does not represent how long you are willing to wait for results. Instead, it represents how long you want the interest operation to persist. For our purposes, we want the interest operation to last until we cancel it, so we used the "timeout never" constant defined by the library.
   - **DOFSystem.InterestOperationListener.** Interest listeners are handled at the DOF level and not covered in this guide. Pass a null argument.

- **Object.** This parameter takes a context argument. As mentioned earlier in this guide, this is an advanced use case not covered in this guide. Pass a null argument.

3. Cancel the interest operation.

   You may explicitly cancel the operation if you no longer need it before you are ready to close the application, but it is likely not necessary in a shutdown routine, where destroying the DOFSystem or the DOF automatically also cancels the interest operation. Note that if you do not need to explicitly cancel the operation, you also do not need to declare or initialize the operation and can simply call the *beginInterest* method.

   Again, the sample does not show this, but the interest operation would likely be canceled in a different block from where it is initiated.

The WaitProvider Query

The method described in this section is a shortcut for creating a full query. It blocks the requestor application and waits until an interest operation has received a provide that matches the query. It then returns a DOFObject for the provider. You would use this call to initialize an object instead of using the method call described in *Creating DOFObject Instances*.

The Java OAL contains two *waitProvider* methods:

- waitProvider(DOFObjectID, DOFInterfaceID, int)
- waitProvider(DOFObjectID, List<DOFInterfaceID>, int)

If you provide a List of multiple interface identifiers, the *waitProvider* method will not return a result unless the provider provides *all* of the interfaces in the query. You must have initiated multiple interest operations beforehand, corresponding to all the interfaces, or an interest operation that uses the wildcard DOFInterfaceID.

Sample Code

The following sample finds a provider of the Time-Based Alarm interface, corresponding with our earlier interest operation:

```
DOFObject provider = system.waitProvider(providerID, TBAInterface.IID, 10000);
```

Code Discussion

The final argument is a timeout. Because this is a blocking call, this timeout represents how long you want to wait for the query to complete. Because the query searches only the local results of interest operations, you do not need to consider network performance in this timeout.

DOFObject instances returned by the *waitProvider* method must still be destroyed, according to the instructions in *Creating DOFObject Instances*.

Customized Query Operations

Creating customized query operations consists of doing the following:

- Building a DOFQuery, which specifies what to query for
- Implementing the DOFSystem.QueryOperationListener interface, which enables you to handle the results of the query

In this section, we'll create a customized query that will prepare us to discover multiple providers in the next section.

Building a Query

You use a builder pattern to create a DOFQuery instance. The DOF OALs commonly use a builder pattern for creating connectivity and other objects, but DOFQuery is the only object covered in this guide that uses the pattern. For DOFQuery, the following steps are involved in following the pattern:

- Instantiate DOFQuery.Builder.
- Use methods of the Builder to set parameters for the DOFQuery. Each of these methods returns a modified version of the Builder instance.
- Call the *build* method of the Builder, which returns a DOFQuery instance that matches the last configuration state of the Builder.

The following parameters can be set for a DOFQuery:

- One or more filters
- One or more restrictions
- A match style

The match style determines how OIDs with attributes will be handled in the query. This guide does not cover using OIDs with attributes and this parameter will be left at its default value in the samples.

Filters

To create a useful query, you must add at least one filter to the builder. Filters determine which provide operations the query will search for. You can create filters that do the following:

- Search for a specific provider (based on OID) of a specific interface (based on IID)
- Search for a specific provider (based on OID) providing multiple specific interfaces (based on multiple IIDs)
- Search for a specific provider (based on OID) providing any interface
- Search for multiple specific providers (based on multiple OIDs) of one or more specific interfaces (based on IIDs)
- Search for any providers of a specific interface (based on IID)
- Search for any providers of multiple specific interfaces (based on multiple IIDs)

When results of a query are found the *interfaceAdded* callback in the query operation listener is called (described later). The following rules apply:

- If the filter contains only a single IID, the *interfaceAdded* callback will be called when a provider of that interface is found; however, if that provider also provides interfaces not included in the filter, the *interfaceAdded* callback will be called again for every interface that provider provides.
- If the filter contains multiple IIDs, the *interfaceAdded* callback will not be called unless a provider is found that provides *all* of the interfaces in the query. If such a provider is found, the *interfaceAdded* callback will be called multiple times (once for each of the interfaces in the filter). In addition, if that provider provides interfaces other than those included in the filter, the *interfaceAdded* callback will also be called for all other interfaces that provider provides.
- If you specify both multiple providers and multiple interfaces in a filter, the results will be for *any* of the specified providers that provide *all* of the specified interfaces.

You can add as many filters as you want. The *interfaceAdded* callback is called whenever results match any single filter. Thus, it may be useful to create multiple filters for the same provider, each with a different single IID or a different set of multiple IIDs.

Note: Remember that you must initiate separate interest operations for all of the providers and interfaces in your query filters.

The signature for the method used to add filters in this guide is *DOFQuery.Builder.addFilter(DOFInterfaceID...)*.

Restrictions

Restrictions enable you to limit the results of a query to only interfaces requested in a filter. For example, if you create a filter for a single IID, and you do not want to receive interface added calls for any additional interfaces that a provider provides, you could also create a restriction for that IID. Restrictions are not used in our samples. We instead ignore non-matching results by checking the IID received in the callback against the IID we are interested in.

Sample Code

The following sample shows how to build a DOFQuery that looks for any providers of the Time-Based Alarm interface:

```
DOFQuery query = new DOFQuery.Builder()
   .addFilter(TBAInterface.IID)
   .build();
```

Implementing a Query Listener

The DOFSystem.QueryOperationListener interface provides the callbacks where you will receive the results of a query operation. The following callbacks are in this interface:

- **interfaceAdded.** If a provider that matches the filter is discovered, this callback is called for every interface that the provider provides, whether or not the interface was included in the filter (unless you have also set restrictions). The common use for this callback is to create DOFObject instances for the found providers, as shown in the sample code.

- **interfaceRemoved.** This callback is called if a found provider cancels one of its provide operations, but the provider still matches one of the filters in the query. You might use this callback for some advanced functionality, but it is not covered in this guide. The sample leaves this method empty.
- **providerRemoved.** This callback is called whenever a provider cancels one of its provide operations and this causes the provider to no longer match any of the query filters. The common use for this callback is to destroy DOFObject instances created in the *interfaceAdded* callback, as shown in the sample code.
- **complete.** This callback is called when the query operation is canceled. The sample leaves it empty.

Sample Code

The following shows an implementation of the DOFSystem.QueryOperationListener that creates objects in the *interfaceAdded* callback and adds them to a Map, then destroys them and removes them from the Map in the *providerRemoved* callback:

```
Map<String, DOFObject> objectMap = new HashMap<String, DOFObject>(2);

private class QueryListener implements DOFSystem.QueryOperationListener{
   @Override
   public void interfaceAdded(Query operation, DOFObjectID oid,
            DOFInterfaceID iid) {
      if(iid.equals(TBAInterface.IID)){
         DOFObject aTBAProvider = system.createObject(oid);
         objectMap.put(oid.toStandardString(), aTBAProvider);
      }
   }

   @Override
   public void interfaceRemoved(Query operation, DOFObjectID oid,
            DOFInterfaceID iid) {
   }

   @Override
   public void providerRemoved(Query operation, DOFObjectID oid) {
      DOFObject aTBAProvider = objectMap.get(oid.toStandardString());
      if(aTBAProvider != null){
         aTBAProvider.destroy();
         objectMap.remove(oid.toStandardString());
      }
   }

   @Override
   public void complete(DOFOperation operation, DOFException exception) {
      objectMap.clear();
   }
}
```

Discovering Multiple Providers

With a query built and a query listener in place that can handle multiple providers, you are nearly ready to initiate a query operation that uses them. However, before you can discover multiple providers, you need to express interest in multiple providers. The interest operation we showed before expressed interest in a single provider. If we used it in combination with our new query, the query would find only the single provider—recall that queries can only find the results of interest operations.

Expressing Interest

To express interest in multiple providers, you could simply call *beginInterest* again for each of those you are interested in. However, you can also pass a multicast address for the DOFObjectID when initiating an interest operation.

Sample Code

The library provides a broadcast address constant, which we used to initiate an interest operation in the following sample:

```
system.beginInterest(DOFObjectID.BROADCAST, TBAInterface.IID,
         DOFInterestLevel.WATCH, DOF.TIMEOUT_NEVER, null, null);
```

It's possible to multicast in other ways, but these are transport-dependent and not covered in this guide.

Initiating a Query Operation

The signatures for the method that initiates the query operation is *beginQuery (DOFQuery, int, DOFSystem.QueryOperationListener, Object)*

Sample Code

The following sample shows the operation (the DOFQuery instance created earlier in this chapter is passed to the *beginQuery* method:

```
DOFOperation.Query queryOp;

...

queryOp = system.beginQuery(query, DOF.TIMEOUT_NEVER, new QueryListener(), null);

...

queryOp.cancel();
```

Code Discussion

The following steps explain the sample code above:

1. Declare a DOFOperation.

Specifically, declare a DOFOperation.Query. The operation needs to be used in multiple blocks.

2. Call the *beginQuery* method.

   By now, you should be fairly familiar with the parameters required. Pass the DOFQuery instance and the operation listeners created earlier.

3. Cancel the query operation.

   You may explicitly cancel the operation if you no longer need it before you are ready to close the application, but it is likely not necessary in a shutdown routine, where destroying the DOFSystem or the DOF automatically also cancels the operation. Note that if you do not need to explicitly cancel the operation in Java, you also do not need to declare or initialize the operation.

Chapter 6: Single-Transaction Asynchronous Operations

The single-transaction asynchronous operations mirror the functionality available in synchronous operations: you can get or set an DOF interface property value and invoke an DOF interface method. The main advantage of these operations for the requestor is that they are non-blocking. Another advantage is that it becomes possible to receive results from multiple providers, where synchronous operations return only the first response received.

Functionality on the provider does not change at all for these asynchronous operations—the same methods implemented for the corresponding synchronous operations respond to the requestor. Consequently, this chapter contains no sample code for the provider.

The samples in this chatper frequently uses a DOFObject variable called *broadcastObject*. This is a broadcast DOFObject created using the broadcast DOFObjectID provided by the library.

Getting Properties Asynchronously

The following diagram shows the communication sequence for an asynchronous get operation:

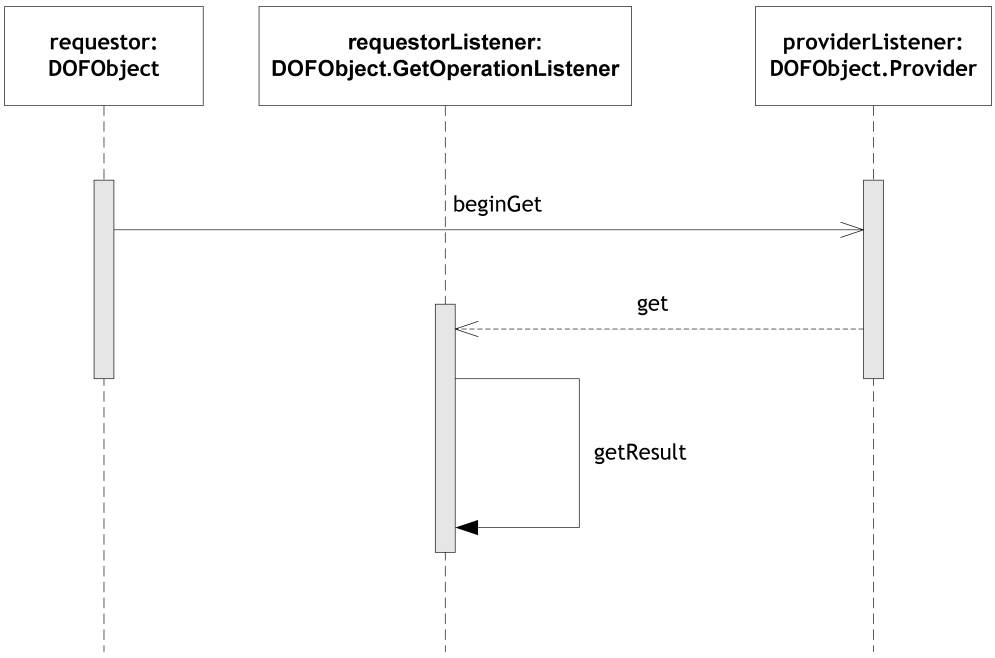

Figure 4. Asynchronous Get Operation Sequence Diagram

Get Operation Listener

As you can see from the sequence diagram, asynchronous get operations require the requestor to implement the DOFObject.GetOperationListener interface to handle the results of the *beginGet* call. This interface has the following callbacks:

- **getResult.** This is called whenever the requestor receives the result of a get operation.
- **complete.** This is called when the asynchronous get operation times out or is canceled. You can use it to clean up any application logic associated with the operation.

Sample Code

The following sample shows an implementation of DOFObject.GetOperationListener that captures the provider's identification and the value of the property and then uses a parent class's method to display the results:

```
DOFOperation.Get activeGetOperation = null;

private class GetListener implements DOFObject.GetOperationListener {
   @Override
   public void getResult(DOFOperation.Get operation,
            DOFProviderInfo providerInfo, DOFValue result,
            DOFException exception) {
      if(exception == null) {
         DOFObjectID providerID = providerInfo.getProviderID();
         String providerIDString = providerID.toStandardString();

         DOFBoolean dofBoolean = (DOFBoolean)result;
         Boolean unwrappedResult = dofBoolean.get();

         parent.displayGetResults(providerIDString, unwrappedResult);
      } else {
         if(exception.getClass().equals(DOFProviderException.class)){
            /* Handle the error. */
         }

         if(exception.getClass().equals(DOFErrorException.class)){
            /* Handle the error. */
      }
   }

   @Override
   public void complete(DOFOperation operation, DOFException ex) {
      if(operation.equals(activeGetOperation)){
         activeGetOperation = null;
      }
   }
}
```

Code Discussion

By now, you should understand the methods shown in the samples. In addition, the library is entirely flexible in allowing you to handle the get results in any way you choose. However, you should note the difference in how exceptions are handled in asynchronous operations. Because you may get multiple results, the Java OAL passes exceptions to you in the callback, rather than throwing them. You must use if blocks or switch cases to handle them, rather than catch blocks.

Calling *beginGet*

The signature for this method is *beginGet(DOFInterface.Property, int, DOFObject.GetOperationListener, Object)*.

Sample Code

The following sample shows this call:

```
public void sendBeginGetRequest(){
   DOFInterface.Property property = TBAInterface.PROPERTY_ALARM_ACTIVE;
   activeGetOperation = broadcastObject.beginGet (property, 10000, new
            GetListener(), null);
}
```

Setting Properties Asynchronously

The following diagram shows the communication sequence for an asynchronous set operation.

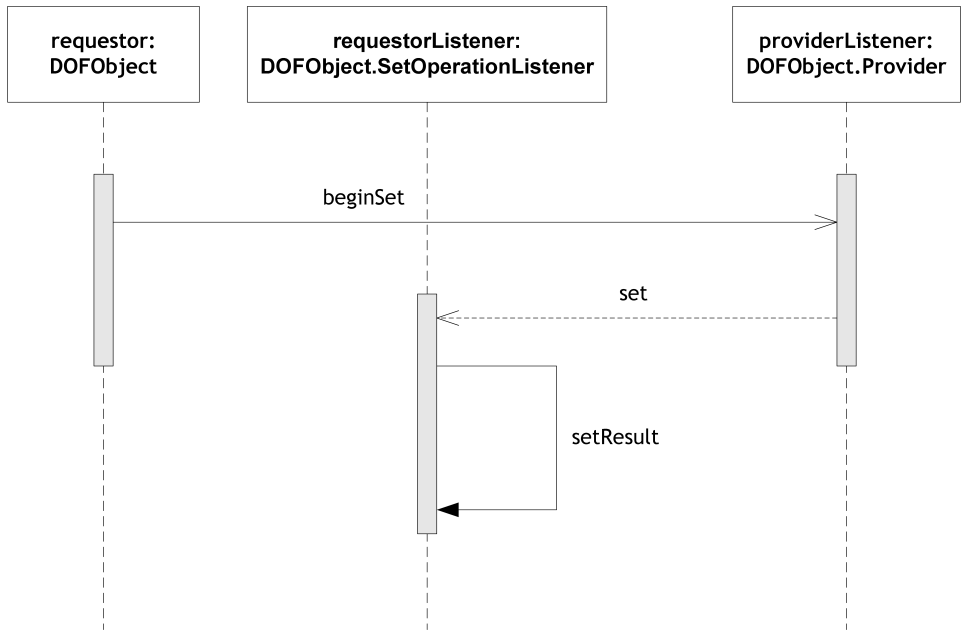

Figure 5. Asynchronous Set Operation Sequence Diagram

Set Operation Listener

Asynchronous set operations require the requestor to implement the DOFObject.SetOperationListener interface to handle the results of the *beginSet* call. This interface has the following callbacks:

- **setResult.** This is called whenever the requestor receives the result of a set operation.
- **complete.** This is called when the asynchronous set operation times out or is canceled. You can use it to clean up any application logic associated with the operation.

Sample Code

The following sample shows an implementation of DOFObject.SetOperationListener that captures the provider's identification and then uses a parent class's method to display the providers whose property values have been set:

```
DOFOperation.Set activeSetOperation = null;

private class SetListener implements DOFObject.SetOperationListener {
   @Override
   public void setResult(DOFOperation.Set operation,
            DOFProviderInfo providerInfo, DOFException exception) {
      if(exception == null) {
         DOFObjectID providerID = providerInfo.getProviderID();
         String providerIDString = providerID.toStandardString();
         parent.displaySetResults(providerIDString);
      } else {
         /* Handle exceptions. */
      }
   }

   @Override
   public void complete(DOFOperation operation, DOFException ex) {
      if(operation.equals(activeSetOperation)){
         activeSetOperation.cancel();
         activeSetOperation = null;
      }
   }
}
```

Calling *beginSet*

The signature for this method is *beginSet(DOFInterface.Property, DOFValue, int, DOFObject.SetOperationListener, Object).*

Sample Code

The following sample shows this call:

```
public void sendBeginSetRequest(boolean active) {
   DOFInterface.Property property = TBAInterface.PROPERTY_ALARM_ACTIVE;
   DOFBoolean setValue = new DOFBoolean(active);
   activeSetOperation = broadcastObject.beginSet (property, setValue, TIMEOUT,
            new SetListener(), null);
```

```
}
```

Invoking Methods Asynchronously

The following diagram shows the communication sequence for an asynchronous invoke operation.

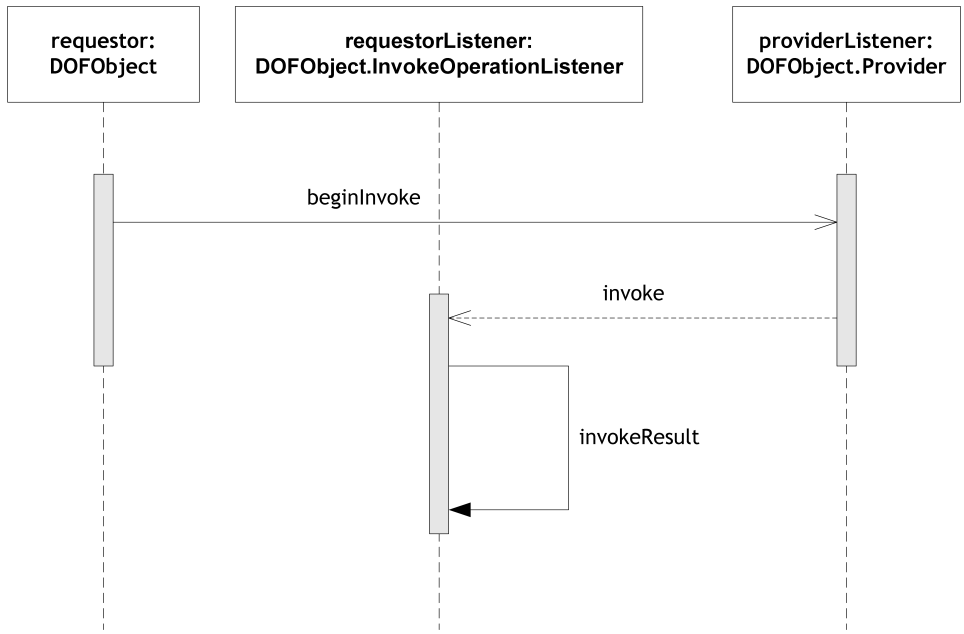

Figure 6. Asynchronous Invoke Operation Sequence Diagram

Invoke Operation Listener

Asynchronous invoke operations require the requestor to implement the DOFObject.InvokeOperationListener interface to handle the results of the *beginInvoke* call. This interface has the following callbacks:

- **invokeResult.** This is called whenever the requestor receives the result of an invoke operation.
- **complete.** This is called when the asynchronous invoke operation times out or is canceled. You can use it to clean up any application logic associated with the operation.

Sample Code

The following sample shows an implementation of DOFObject.InvokeOperationListener that captures the provider's identification and the return parameter and then uses a parent class's method to display the results:

```
DOFOperation.Invoke activeInvokeOperation = null;

private class InvokeListener implements DOFObject.InvokeOperationListener {
   @Override
   public void invokeResult(DOFOperation.Invoke operation,
            DOFProviderInfo providerInfo, List<DOFValue> result,
            DOFException exception) {
      if(exception == null) {
```

```
            DOFObjectID providerID = providerInfo.getProviderID();
            String providerIDString = providerID.toStandardString();
            DOFValue value = result.get(0);
            DOFBoolean dofBoolean = (DOFBoolean)value;
            Boolean unwrappedResult = dofBoolean.get();
            parent.displayInvokeResults(providerIDString, unwrappedResult);
         } else {
            /* Handle exceptions. */
         }
      }

      @Override
      public void complete(DOFOperation operation, DOFException ex) {
         if(operation.equals(activeInvokeOperation)){
            activeInvokeOperation.cancel();
            activeInvokeOperation = null;
         }
      }
}
```

Calling *beginInvoke*

The signatures for this method is *beginInvoke(DOFInterface.Method, int, DOFObject.InvokeOperationListener, Object, DOFValue...)*.

Sample Code

The following sample shows this call:

```
public void sendBeginInvokeRequest(Date alarmTime) {
   List<DOFValue> parameters = new ArrayList<DOFValue>();
   DOFDateTime inputParameter = new DOFDateTime(alarmTime);
   parameters.add(inputParameter);
   DOFInterface.Method method = TBAInterface.METHOD_SET_NEW_TIME;
   activeInvokeOperation = broadcastObject.beginInvoke (method, parameters,
            TIMEOUT, new InvokeListener(), null);
}
```

Chapter 7: Ongoing Asynchronous Operations

In this chapter, we'll discuss the two ongoing asynchronous operations that deal directly with items of interface functionality: subscribing to an DOF interface property and registering for an DOF interface event. You will learn how to implement these operations on the provider and how to implement operation listeners and initiate the operation with the requestor.

Subscribing to a Property

The following diagram shows the communication sequence for a subscribe operation.

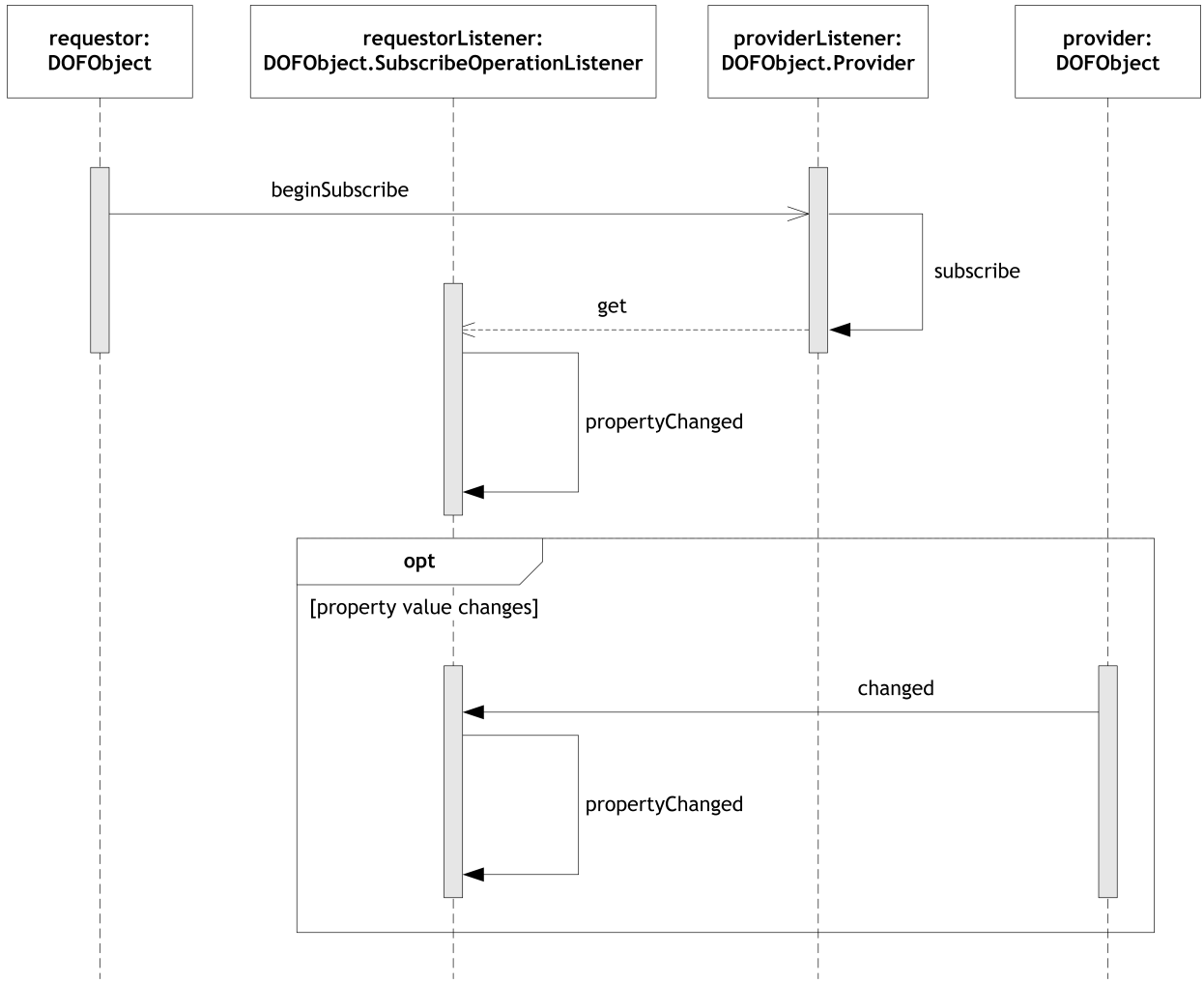

Figure 7. Subscribe Operation Sequence Diagram

Providers

If *all* the following conditions are true, the provider does not need to do any additional work to implement subscribe operations:

- You have already implemented the *get* callback for the property. As you can see from the sequence diagram, it is the *get* callback that sends the requestor the initial value of the property.
- You are extending the DefaultProvider class (meaning the *subscribe* callback from the Provider interface has been implemented). The default behavior is to automatically accept subscribe requests.

  The *subscribe* method is for the provider's internal use. For example, you could use it if you want the provider to track the requestors that are subscribed to a property. If you do override this method, you must ensure that you call *DOFRequest.respond*. Returning an exception in the *respond* method causes the requestor's subscription to fail. Samples of the provider's *subscribe* callback are not shown in this guide.

- There is no way for the property value to change other than through set operations from requestors. The library automatically notifies subscribed requestors if the property value changes through a set operation. If, however, the property can be changed by the *invoke* callback or you have internal application logic that can cause a property value to change, you must call the *changed* method.

Sample Code

The following sample shows the *DOFObject.changed* method:

```
public void setActive(boolean active){
   if(isActive != active){
      isActive = active;
      DOFInterface.Property property = TBAInterface.PROPERTY_ALARM_ACTIVE;
      provider.changed(property);
   }
}
```

Subscribe Operation Listener

Subscribe operations require the requestor to implement the DOFSubscription.Listener interface to handle the results of subscriptions call. This interface has the following methods:

- **propertyChanged.** This is called whenever the provider sends notification that a property value has changed. It is also called immediately after initiating the operation when the provider sends the current property value.
- **stateChanged.** This is called when the state of a subscription has changed (become active or inactive).
- **removed.** This is called when the subscription listener is removed. You can use it to clean up any application logic associated with the operation. The sample leaves this method empty.

Sample Code

The following sample shows an implementation of DOFSubscription.Listener that captures the provider's identification and the new property value and then uses a parent

class's method to display the results:

```
private class SubscribeListener implements DOFSubscription.Listener {
   @Override
   public void propertyChanged(DOFSubscription subscription,
            DOFObjectID providerID, DOFValue value) {
   String providerIDString = providerID.toStandardString();

   DOFBoolean dofBoolean = (DOFBoolean)value;
   Boolean unwrappedResult = dofBoolean.get();

   parent.displaySubscribeResults(providerIDString, unwrappedResult);
   }

   @Override
   public void stateChanged(DOFSubscription subscription,
            DOFSubscription.State state){
   }

   @Override
   public void removed(DOFSubscription subscription, DOFException exception) {
   }
}
```

Instantiating DOFSubscription

When you instantiate DOFSubscription, it automatically attempts to establish a subscription. The signature for the constructor is *DOFSubscription(DOFSystem, DOFObjectID, DOFInterface.Property, int, Listener, Object)*.

Sample Code

The following sample shows the *DOFSubscription* constructor:

```
public void sendBeginSubscribeRequest(){
   try {
      subscriber = new DOFSubscription(system, currentProvider.getObjectID(),
               TBAInterface.PROPERTY_ALARM_ACTIVE, MIN_PERIOD,
               new SubscribeListener(), null);
   } catch (DOFErrorException e) {
      /* Handle the error. */
   }
}
```

Code Discussion

A couple of parameters deserve consideration:

- **DOFSystem.** The DOFSubscription class needs a DOFSystem instance to create its own DOFObject instances.
- **DOFObjectID.** This must be the OID of the provider whose property you wish to subscribe to.
- **int.** This is the minimum period parameter. It represents the interval that you want between property changed notifications for the same provider and property.

This means that if the property changes more frequently, you will be notified of the most recent change only after the minimum period has expired.

Registering for an Event

The following diagram shows the communication sequence for a register operation.

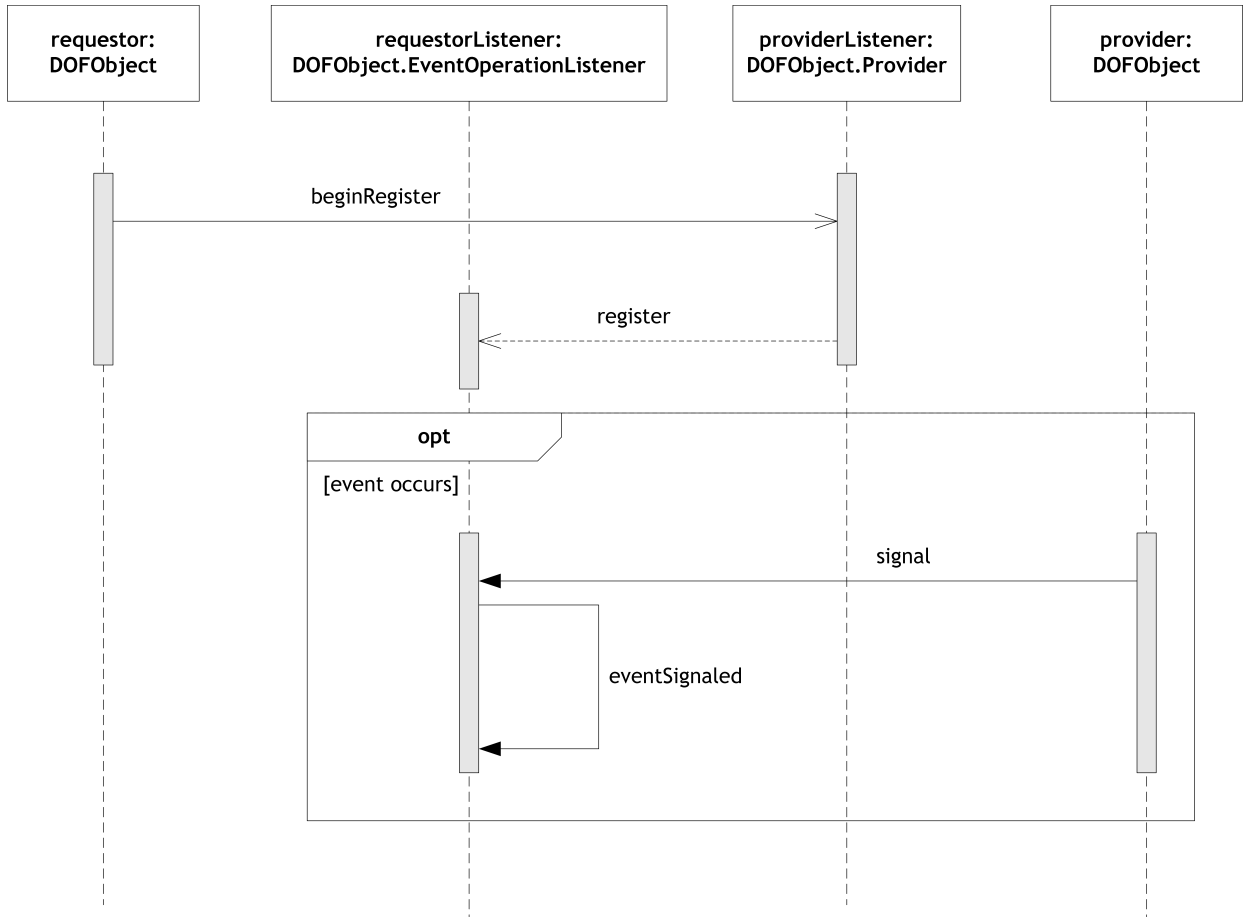

Figure 8. Register Operation Sequence Diagram

Providers

Implementing the *register* method is optional for the provider. Like the *subscribe* method, it is for the provider's internal use. If you do implement it, the same rules apply for calling *DOFRequest.respond* that apply to the *subscribe* method.

However, unlike the *changed* method for subscriptions, which the provider need only call under specific conditions, the *signal* method must always be called to enable event notification.

Sample Code

The following sample shows the *DOFObject.signal* method:

```
public void checkForEvent(){
   if(alarmTime != null){
      if(alarmTime.compareTo(now) <= 0 ){
```

```
            if(isActive){
               DOFInterface.Event event = DOFInterface.EVENT_ALARM_TRIGGERED;
               if(provider.isSignalRequired(event))
                  provider.signal(event);
                  alarmTime = null;
            }
         }
      }
   }
```

Code Discussion

Follow these steps for using the *signal* method:

1. Check for the conditions that should trigger the event.
2. Call *isSignalRequired* to determine whether any requestors are registered for the event.
3. Package the output parameters.
4. Call the *signal* method.

Register Operation Listener

Register operations require the requestor to implement the DOFRegistration.Listener interface to handle the registration. This interface has the following methods:

- **eventSignaled.** This is called when an event is triggered. It is also called immediately after initiating the operation, when the provider sends an initial acknowledgment.
- **stateChanged.** This is called when the state of a registration has changed (become active or inactive).
- **removed.** This is called when the registration listener is removed. You can use it to clean up any application logic associated with the operation. The sample leaves this method empty.

Sample Code

The following sample shows an implementation of DOFRegistration.Listener that captures the provider's identification and the new property value and then uses a parent class's method to display the results:

```
private class RegistrationListener implements DOFRegistration.Listener {
   @Override
   public void eventSignaled(DOFRegistration registration,
            org.opendof.core.oal.DOFObjectID providerID,
            java.util.List<org.opendof.core.oal.DOFValue> values) {
      String providerIDString = providerID.toStandardString();
      String message;

      if(values == null){
         message = "ack";
      } else {
         message = "event triggered";
      }

      parent.displayEventResults(providerIDString, message);
```

```
    }

    @Override
    public void stateChanged(DOFRegistration registration, DOFRegistration.State
             state) {
    }

    @Override
    public void removed(DOFRegistration registration,
             org.opendof.core.oal.DOFException exception) {
    }
}
```

Code Discussion

The *eventSignaled* callback is initially called when the provider acknowledges the registration, which does not mean the event was triggered. It is important that code in this callback distinguish between the initial acknowledgement and subsequent calls when the event has been triggered:

- When an event was triggered, the DOFValue List argument will always be filled. If the event has no output parameters, the List will be empty, but not null.
- For an acknowledgement, the List argument will be null.

Instantiating DOFRegistration

When you instantiate DOFRegistration, it automatically attempts to establish a registration. The signature for the constructor is *DOFRegistration(DOFSystem, DOFObjectID, DOFInterface.Event, Listener, Object)*.

Sample Code

The following sample shows the *DOFRegistration* constructor:

```
public void sendBeginRegisterRequest(){
    try{
       registrant = new DOFRegistration(system, currentProvider.getObjectID(),
                TBAInterface.EVENT_ALARM_TRIGGERED, new RegistrationListener(),
                null);
    } catch (DOFException ex){
       /* Handle the exception */
    }
}
```

Chapter 8: Activate-Level Interest

Earlier in this guide, we discussed how a requestor can perform interest and query operations to discover providers, and we introduced the concept of activate-level interest. To review, activate-level interest is a request to begin providing an interface. Up to this point in the guide, our provider has performed a provide operation when it initializes. This chapter covers how to program a provider to instead begin dynamically providing an interface in response to activate-level interest.

To respond to activate-level interest, the provider must implement the DOFSystem.ActivateInterestListener interface. The library then notifies the system when an activate- or higher level interest is received. This interface has the following methods:

- **removed.** The library calls this function when the listener is removed or the system is destroyed. You can use it to free resources associated with the listener; however, since our sample code does not allocate any resources for the listener, we leave this method empty to ignore removal of the listener.
- **activate.** The library calls this method when an activate-level interest is received.
- **cancelActivate.** The library calls this method when an interest request is canceled or times out.

This chapter will show how to implement the interface. After implementing it, you would add it to a DOFSystem instance using the *DOFSystem.addActivateInterestListener* method.

Writing the *activate* Method

Use the *activate* method of the DOFSystem.ActivateInterestListener interface to determine whether the interest applies to your provider and to begin providing the interface if it does.

Sample Code

The following sample shows an implementation of the *activate* method:

```
@Override
public void activate(DOFSystem system, DOFRequest request, DOFObjectID oid,
         DOFInterfaceID iid) {
   if(oid.equals(DOFObjectID.BROADCAST) || oid.equals(providerID)) {
      if(iid.equals(TBAInterface.IID)) {
         synchronized(this) {
            interestedCount++;
            if(provider == null) {
               provider = system.createObject(oid);
               provideOperation = provider.beginProvide(TBAInterface.DEF,
                        DOF.TIMEOUT_NEVER, new ProviderListener(), null);
            }
         }
      }
   }
}
```

Code Discussion

The sample code does the following:

- Examines the OID in the interest to determine whether the interest applies to the provider. If the OID is the broadcast OID or the provider's specific OID, the interest can be assumed to apply to the provider.
- Examines the IID in the interest. If the OID is the broadcast OID, this step further helps to determine whether the interest applies to the provider. In addition, if the provider were capable of providing multiple interfaces, this step would be vital to determining which interface to provide in response to the interest.
- Keeps and increments a count of interest operations received for the interface. This is important so that the provider does not stop providing an interface if multiple requestors have expressed interest. We'll discuss this further in the next section.
- Creates a DOFObject. This code assumes that the DOFObject was declared and set to null in a class field with the variable name *provider*. The DOFObject instance is then dynamically created in response to the interest. This is a reasonable use case, but you might also wish to create DOFObject instances elsewhere.
- Checks to see whether a provide operation (a previously declared DOFOperation.Provide) for the interface is null. If not, it begins providing the interface.

Writing the *cancelActivate* Method

Use the *cancelActivate* method of the DOFSystem.ActivateInterestListener interface to be notified when interest is canceled and stop providing.

Sample Code

The following sample shows an implementation of the *cancelActivate* method:

```
@Override
public void cancelActivate(DOFSystem system, DOFRequest request, DOFObjectID oid,
         DOFInterfaceID iid) {
   if(oid.equals(providerID)){
      if(iid.equals(TBAInterface.IID)){
         synchronized(this) {
            interestedCount--;
            if(interestedCount <= 0){
               provideOperation.cancel();
               provider.destroy();
               provider = null;
            }
         }
      }
   }
}
```

Code Discussion

Like the *activate* method samples, this sample compares the OID and IID received to a known OID and IID to determine the binding the interest applies to. Notice that it decrements the interest count and does not stop providing until the count is zero or less.

Appendix A: Formatting OIDs

OIDs consist of both an OID class and data. The OID class must be a valid DOF OID class that has been formally registered, and the data is a number or string appropriate for identifying data in that class.

OIDs must be globally unique, and in a production environment, they should be carefully managed. However, this appendix covers how you can create OIDs, using three classes that have already been registered, to use in experimentation and testing:

- GUID
- Domain
- Email

Note: All of these classes create large OIDs and may not be ultimately appropriate for resource-constrained devices. In addition, to create domain- or email-class OIDs, even for testing, you must own a properly registered Internet domain that meets Internet Engineering Task Force (IETF) specifications.

GUID Class OIDs

To create GUIDs that you may use as the data portion in OIDs requires only a tool that can randomly generate a 16-byte GUID in hexadecimal. Free browser-based GUID generators are readily available online.

Of the options outlined in this appendix, the GUID class has the advantage of being potentially the shortest type of OID, as well as being freely available (not requiring that you own an Internet domain). Its potential disadvantage is that GUIDs are not as readily human-readable as domain names and email addresses, so that, in experimentation and testing, it may be easier to keep track of the objects being identified in your code when you use the other classes discussed in this appendix.

Domain Class OIDs

With your own registered Internet domain, you can create an extremely large number of domain class OIDs. The requirement for domain registry is to ensure that OIDs remain globally unique, although the OIDs do not actually use DNS resolution. For this reason, once you have a registered domain name, you can create multiple OIDs using any subdomain or other format that conforms to the IETF's RFC 2822, whether or not the resulting URL can actually be resolved by a DNS server. The only requirement is that you take responsibility for tracking the OID's uniqueness within your own domain. So, for example, if you owned "domain.com," you could use any of the following as OID data, even if you have not set up DNS resolution for them:

- domain.com
- requestor.domain.com
- provider.domain.com

So as long as you own the domain, and you manage unique subdomain names, the number of OIDs you can create is limited only by the maximum length of 63 bytes, the characters allowed by RFC 2822, and the portion of your OID data that is the fixed name of your registered domain.

Email Class OIDs

The email class extends the number of OIDs you can create using the same registered domain used in your domain class OIDs. You must format emails according to specifications outlined in RFC 2822.

As with the domain class, you may use subdomains that are not resolved by a DNS server. This provides options for managing uniqueness. You can use email classes with OIDs where the text string is different, but the domain is the same, as in the following examples:

- requestor@domain.com
- provider@domain.com

You can also create OIDs that share the same text string, but where subdomains are different, as in the following examples:

- object1@requestor.domain.com
- object1@provider.domain.com

In addition, for experimental and testing purposes only, it is reasonably safe to use your own email address or addresses as OIDs, even if you don't own their associated domains.

Note: The maximum length of an email class OID is also 63 bytes.

Formatting OIDs for DOFObjectID Create Methods

After choosing the data you want to use in your OIDs, you need to know how to format the data to pass it into a DOFObjectID create method.

You can use the *DOFObjectID.create(String)* method to create all three classes of OID discussed in this appendix. However, the string passed to this method must be in the standard string format that the ENC has defined for OIDs.

To use the standard string format, you will first need to obtain the identifier that has been assigned to the class.

The identifiers for the classes discussed in this appendix are the following:

- GUID: 128
- Domain: 6
- Email: 3

The basic rules for the standard string format are as follows:

1. The string must be enclosed in square braces ([]).
2. Within the square braces the format is <class identifier>:<OID data>.
3. Hexadecimal values in the OID data must be enclosed in curly braces ({}).
4. The string should not contain any spaces.

Code Samples

The following is an example of how to create a DOFObjectID using a GUID (in hexadecimal) as the OID data:

```
DOFObjectID providerID = DOFObjectID.create
         ("[128:{050234706bb1453289d9461055cef60c}]");
```

The following is an example of how to create a domain class DOFObjectID:

```
DOFObjectID providerID = DOFObjectID.create("[6:provider.opendof.org]");
```

The following is an example of how to create an email class DOFObjectID:

```
DOFObjectID providerID = DOFObjectID.create("[3:provider@opendof.org]");
```

Summary

The GUID, domain, and email OID classes provide nearly limitless possibilities for creating unique OIDs, which you can use fairly freely in experimentation and testing. However, remember that in a production environment, it is vital to manage and monitor the uniqueness of OIDs. In addition, every OID class has its own specifications for use. System designers should carefully plan which OIDs will be used in a production environment.